Object Oriented Programming
With
Visual Basic .NET

Jinan Darwiche
## Contents

**Chapter 1 Object Oriented Programming Theory** ........................................... 1  
1.1 What is an object? ......................................................................................... 1  
1.2 OOP Claim to Fame ..................................................................................... 1  
1.3 Properties ..................................................................................................... 2  
1.4 Events .......................................................................................................... 2  
1.5 Methods ........................................................................................................ 2  
1.6 What’s in it for me? ....................................................................................... 2  
1.7 Inheritance ................................................................................................ 3  
1.8 Encapsulation ............................................................................................... 3  
**Exercises** ....................................................................................................... 4  

**Chapter 2 VB Classes and Objects** ................................................................. 5  
2.1 An Employee Example ................................................................................ 5  
2.2 My First OOP Project ................................................................................... 6  
\hspace{1em} 2.2.1 My First Property ........................................................................... 7  
\hspace{1em} 2.2.2 My First Object .............................................................................. 8  
2.3 Going Public or Staying Private .................................................................. 12  
2.4 My First Method .......................................................................................... 13  
2.5 ReadOnly and WriteOnly Properties ............................................................ 16  
2.6 Default Properties ...................................................................................... 18  
\hspace{1em} 2.6.1 Accessing Default Properties ......................................................... 19  
**Exercises** ....................................................................................................... 19  

**Chapter 3 Constructors** .................................................................................. 21  
3.1 Constructor method ...................................................................................... 21  
3.2 ReadOnly and Sub New ................................................................................ 23  
3.3 Tying it together - General Design Issues .................................................. 23  
3.4 Private and Public Methods ........................................................................ 24  
3.5 Is Nothing .................................................................................................. 25  
3.6 Killing Objects ............................................................................................ 26  
**Exercises** ....................................................................................................... 26  

**Chapter 4 Events** .......................................................................................... 27  
4.1 Event Declaration ......................................................................................... 28  
4.2 Event Firing .................................................................................................. 28  
4.3 Connecting Events with Event Handlers .................................................... 29  
\hspace{1em} 4.3.1 Static Event Handling ...................................................................... 30  
\hspace{1em} 4.3.2 The Handles Clause and Static Handling: ........................................ 32  
4.4 Dynamic Event Handling Using AddHandler .......................................... 34  
\hspace{1em} 4.4.1 Handling events using Add Handler ................................................. 34  
\hspace{1em} 4.4.2 Using RemoveHandler to Stop Handling Events ........................... 35  
\hspace{1em} 4.4.3 Adding Handlers without Removing Them ...................................... 35  
4.5 Sequence of Events ..................................................................................... 36
Chapter 9 Collections ........................................................................................................ 86
  9.1 Collections vs. Arrays of Objects ............................................................................. 86
  9.2 Creating CollectionBase Classes .......................................................................... 87
  9.3 Predefined Collections ......................................................................................... 94
  9.4 One-Based Collection and Zero-Based CollectionBase ........................................ 95
  9.5 Using the Collection Object .................................................................................. 95
  9.6 Inside a Collection Object ..................................................................................... 96
  Exercises ......................................................................................................................... 97

Chapter 10 Stacks, Queues, and Lists .............................................................................. 98

Chapter 11 Multi-Form Applications ............................................................................. 104
  11.1 Windows Multi-Forms Projects ............................................................................ 107
  11.2 Creating an MDI Project ....................................................................................... 107
  11.3 Controlling MDI Behavior ................................................................................... 108
  11.4 Working with Window to Control Child Forms .................................................. 109
  11.5 Child Menus .......................................................................................................... 110
  11.6 Working with DialogBoxes ................................................................................... 111
  Exercises ......................................................................................................................... 112

Chapter 12 Deployment and Assemblies ....................................................................... 113
  12.1 Deploying Applications and Components ............................................................. 113
  12.3 Assemblies ............................................................................................................. 116
    12.3.1 Creating Assemblies ....................................................................................... 117
  12.4 The Global Assembly Cache ................................................................................ 118
  12.5 Strong Names ......................................................................................................... 118
  12.6 Installing an Assembly in the GAC ...................................................................... 118
  12.7 The Common Type System and Closing Remarks ............................................... 120
Preface

Who this book is intended for
Programmers with good knowledge of the Visual Basic programming language, and programming in general, should find this book helpful in learning Object Oriented Programming concepts, and how to apply these concepts in creating classes and objects in Visual Basic .NET. Programmers with experience not related to Visual Basic should also find the book useful, but need to learn about the basic syntax of the language. I recommend using www.msdn.microsoft.com and look for the Visual Basic .NET reference.

Compiler Versions
This book covers programming concepts in Visual Basic .NET. Although in some cases (creating menus, and Deployment applications), the syntax is specific to the 2008 version of the compiler, however, in general the code shown applies to all versions of the VB .NET compiler.

Additional material online
Check out the author’s homepage for accompanying material such as the errata sheet and additional code samples. The author’s homepage is: http://homepage.smc.edu/darwiche_jinan

How this Book is Organized
The information presented in this book is sequential in its coverage. Hence you must cover chapter 1 before you can learn and practice the material of chapter 2. This applies to chapters 1 through 9. Chapters 10-12 are independent, and may be covered in any order.

Throughout the book, important concepts are marked with “Note” and code samples are indented. Keywords are shown in bold and italics.

About the Author
Jinan Darwiche holds a Bachelor of Science degree in Electrical Engineering, and a Masters in Computer Science. She has worked for 8 years in different positions in the Computer Science field before joining Santa Monica College in 2000, where she is currently a faculty in the Computer Science and Information Systems department. She teaches programming at different levels, Project Management and Website development courses.
Chapter 1 Object Oriented Programming Theory

Object Oriented Programming, OOP, has become so popular due to its ability to allow programmers to reuse code. This concept of code reuse is very powerful because it saves so much effort in writing code, and more importantly it allows someone with little programming knowledge to create professionally looking applications that connect to databases and run over the internet.

At its heart, Object Oriented Programming is a data representation concept that has evolved over the years. Our aim in this book is to learn the basics of OOP, and how Visual Basic represents and manipulates data in OOP.

1.1 What is an object?

OOP claims to be able to represent any data item into an object. Therefore, an object can be anything we need it to be, for example, a car, house, or pet, and even you!

To detail how we represent an object in programming languages that are OOP-based, it is best to study an example. We already stated that a car could be an object, so let's look at how we humans create cars, and then use that example to examine how we can represent that same car in OOP.

A car factory is where cars are born. To simplify, a car that reaches a dealership goes through these steps:

1. A car designer draws the shape design and the engine and other mechanical parts of the car. In reality, many engineers work on that, but we'll pretend that these are one person as a concept.
2. After many steps of redesigning and verifications of the design, a 'mold' of the car parts is made. The mold is used to create many 'copies' of the car. At the factory's assembly line, many cars are generated.
3. Even though all the cars of that design share the same basic features, however they are different in many other aspects, like the exterior color, type of seats (leather vs. cloth), etc.

In OOP, we say the car design is a class definition, while the cars born in the factory are objects. Hence we have one class based on which as many objects as needed are produced where each object, an actual car, has the general properties that the class design specifies, like number of doors, number of cylinders, color of exterior, etc, however each particular car has its own value of number of doors, cylinders, color, etc.

Before we continue with more concepts, it is important at this point to list the advantages of using OOP.

1.2 OOP Claim to Fame

In OOP, to represent data like your car, you start by representing a general concept of a car. Hence you specify what a car looks like, called properties in OOP, and how it should be used or behave, called methods. Then you create the actual car with the color, engine, and other qualities of your particular car, a process we call instantiation. This mechanism, of creating a general design feature, then out of that design you can 'manufacture' hundreds of cars, is very
Chapter 1  

Object Oriented Programming Theory

powerful, because it allows you to produce many cars without having to redesign each one separately. If cars are created through code, then instantiation is the mother of code re-use. The concept of code re-use goes further in OOP through inheritance, through which a child inherits its parent, or from programming viewpoint, a child code inherits its parent code, hence the child code does not repeat the parent code, but can further contain more properties and methods, and or improvements on them.

First, let's detail through, what a class constitutes, and then we'll go over what inheritance is and why it reflects code reuse at its best.

1.3 Properties

A property is an attribute that describes what an object may look like. For example, a car’s color, number of doors, VIN number, or any item that may have a value is a property. Property values may be assigned or evaluated through code. For example, you may assign the car’s color to black, and later in code we may need to display what the color is. The importance of assigning vs. evaluating what a property value is will become clearer later as we work in VB code in particular.

Properties are not enough to represent a car. For example, a car is driven, washed, serviced, runs out of gas, gets pumped with gas, with many other actions that we can list. These actions can also be represented in OOP, using methods and events.

1.4 Events

As a VB programmer, you have been using events when you write code for controls’ event handlers. Recall that in a Windows application, if you double click a Button control, for example, then the Integrated Development Environment, IDE, takes you to the code view of the form, and creates the event handler signature (Private Sub ...) and the End Sub lines. All you need to do is write the code in between, and that code runs whenever the user does an action causing the event to ‘happen’ (we use the word fire in place of ‘happen’ in OOP). So you are not a stranger to events, but as a programmer of Windows applications, you have been handling events only. In this book, you will learn how to initiate, or fire, events.

To continue with our car example, let’s assume that other properties the car has are the gas level, and the gas tank’s capacity. The amount of gas in the tank can be referred to as the gas level property. When the gas level value reaches a certain minimum, an indicator turns on, meaning that you have a certain minimum amount of gas that is about to run out. The action of the low gas light indicator turning on is not a property, but an event in OOP. Events are actions that occur due to a property or a number of properties changing their value. Events are of extreme importance in programming, as we shall see later once we discuss how VB implements OOP.

1.5 Methods

Other than properties and events, we also need to represent how to use a car. For example, driving a car causes the gas level property as well as a mileage property to change. We agree though that driving is neither a property nor it is an event. It is an action that may cause one or more property to change, and it may also cause events to happen, like the gas level becoming too low and the low gas indicator turning on.

1.6 What’s in it for me?

As a programmer, you may not see the benefits of OOP yet. An example is due at this time. While programming Windows applications, you have been using forms and adding controls to forms all the time. What do you think a control is? A control is a graphical representation of code
--a Class code- that some other programmer has created, and then other programmers (users of the VB IDE) use that code to create multiple instances --objects- of that class and then run that code during the runtime. Does the word "multiple" sound familiar? Yes, those textboxes and other controls that you have been adding from the toolbox onto a form so easily, are nothing but a class that has been created by someone else, and then you create instances of these classes so effortlessly. We will not discuss the visual aspect of this effortlessly, but only focus on the multiple instances aspect.

Now that we realize that without OOP, creating those controls would have been an enormous task, we are beginning to appreciate OOP more. What if someone tells you that not only can you create add and use those controls that easily using classes and objects, but you can also improve on their design, such as making a textbox oval in shape? You will even love OOP more, and wonder how we can improve on existing controls or classes.

1.7 Inheritance

Using our example of a car, when someone creates a car design, then if someone else likes that car but wants to modify it to improve its performance, then one can produce an improved car. However, the improvements require access to the original design, but the original design is kept in secret drawers somewhere, where access to it is prohibited unless authorized by law. Not in OOP, where a class may be used to produce a better version, and then objects based on the better version can be produced. This process of inheritance can be done with or without direct access to the original class code, as we shall see later.

This concept of re-using a class to create an improved class is what makes OOP powerful. This code reuse feature has many aspects but we will cover two of those, namely, Inheritance and polymorphism.

Inheritance is the bases of OOP’s popularity. You can use a class such as a car and make additional improvements; you’re not stuck with the original design, nor do you have to start over if you want to get, say a car that drives at a better Miles per Gallon, MPG. With Inheritance, you can use a previously designed car as a base, and then indicate what the new qualities to add are and what existing qualities or behaviors to improve.

A programming example that uses inheritance would be when you use a Textbox class as a base class and then inherit that class to create say an Oval Textbox class, hence changing its original shape property into an improved oval shape property. This ability to use an existing code to produce a better class is a great example of code re-use, and you don’t need the source code of the Textbox class, as the compiled version is sufficient.

To recap, so far we have covered the basic concepts of:

- OOP is about Code Re-use.
- Classes are designs or blueprints and objects are copies or instances of classes.
- A class design is done by specifying the properties, methods, and events an object may have.
- Inheritance allows a class to be improved without needing its source code, another great example of code re-use.

1.8 Encapsulation

This is another great aspect of OOP that makes it attractive for beginner programmers to enjoy code re-use and create sophisticated applications. Encapsulation refers to when a class creator hides so many complicated details of the class code from the programmer who will be using that class to create objects. This detail-hiding feature, makes coding on the object side easy, and protects the original class code from destruction by the programmer creating objects.
To demonstrate the importance of detail-hiding, called **encapsulation** in OOP, a good parallel to use in this case is the television set. Imagine a TV without a button or remote to increase or decrease the sound volume. Instead you would have to access the internal electronics of the set to manipulate the sound each time you needed to. In that case, the details of the set are exposed, and that may be dangerous to the user, and may cause the television to be destroyed.

The concept of encapsulating the electronics to manipulate the sound volume is important for two reasons:

1) It protects the inner parts of the television from being destroyed by the user. Hence if the TV was a class, then we are protecting the class code.

2) It makes the television set easier to use and hence the user creating objects will find using those objects easy by not having to worry about so many mechanics and details.

It is now time to start coding in VB. In the next chapter we will apply the basic concepts of this chapter into programming in VB .NET.

**Exercises**

1.1 List the advantages of OOP?
1.2 What are the differences and similarities between classes and objects?
1.3 Would you consider a form to be an object or a class? Perhaps providing examples will help.
1.4 Provide an example of inheritance in a programming concept? How does this example demonstrate code reuse?
Chapter 2 VB Classes and Objects

Now that we learned about the basic conceptual design of an object-oriented class, it is time to start implementing the code.

Before we proceed, you need to know that in a typical OOP project, there may be a component of the software made of Classes, another component made of code that creates objects based on these classes. It is a typical situation that the programmers of the class code not to be the same programmers working on the object code. We refer to the class code as the source, or server, while the code that creates objects and manipulates them is referred to as the client code. Also worthwhile mentioning here is that it is typical for the object code, not to have access to the original server source code, but only the compiled class code which is usually in an executable file referred to as the dynamic link library – dll. You will see later why this is significant. For this chapter, you will write the class source code, and you will use this source code in the same project that creates objects out of that class.

From a design perspective, it is important to write each component – classes and objects- to be independent from each other. For example, a typical client code would be a form or multiple forms. The class code should not refer to any controls on the form, as the class code should work for any client project type, hence no reference to any form name or controls should be made from the class code.

2.1 An Employee Example

Let’s take an employee as an example. We will create a class code in VB, then using a form we will learn how to instantiate objects based on that class.

First, think about what properties, methods and events an Employee class must have, and then code those items. You may probably produce a similar design to the following:

Properties
- First Name
- Last Name
- Date of Birth (DOB)
- Social Security Number (SSN)
- Department
- Salary
- Date of Hire (DOH)
- Days in Current Position (DCP)
- Rank, which is a number from 1-10, with 10 being the highest rank that equates with a higher salary.

There may be other properties that you can think of, but we’ll just use those for now.

Methods
- Recall that a method represents actions that can affect the property values. Examples could be:
  - Promote, which allows us to increase the salary based on the rank, DCP.
  - UpRank, which causes the rank to increase based on DOH, DCP.

Events
As stated before, events are ways to communicate that an action has occurred that may require a reaction. We’ll do more on Events in chapter 4. In this case, we may have: InitializePromote, which fires when the DCP is > 475 SalaryAboveMillion, which fires when the Salary value is >= 1,000,000.

Again, this is a preliminary design that you can edit as you see fit. We will keep it simple so that we can focus on the code writing process.

2.2 My First OOP Project

Using the VB IDE, start a new Windows project. The form GUI and code will be used to implement an object (client code), while the class code will be done in a component added under the Project Add menu. We will start by creating the Employee class. For future projects, if your goal is to develop only the class code and not the client code, you will choose Class Library project type. For deployment of class codes onto other computers, we will learn about how to create setup projects towards the end of this book. For now choose a Windows Forms project then add a class component as shown below.

1. After creating a new Windows project, click the Project menu then Add Class.
2. A class is like a module; it will have one file to hold its code, and it does not have a GUI. Type Employee for the name of the file then click Add.

3. You will now be inside the code view with the words “Class Employee” at the top. The word that follows the keyword Class is the class name, and that will be the same as the filename you choose in step 2.
You are now ready to type the code to create the rest of the class.

2.2.1 My First Property

The code needed to implement the FirstName property is shown below. A property is typically implemented by a property procedure (a special type of Sub Procedure) and a privately declared variable.
End Get
Set (byVal Value As String)
    If Value <> Nothing Then 'we will learn what this check is later
        mFirstName = Value
    End If
End Set
End Property

To understand how the code works, keep the big picture in mind: A class is nothing but the blueprint of an object. Without the object, the class is like a car design - you cannot drive it! Seeing how an object uses the code above helps you understand the code components. Let’s write the code to create the object.

2.2.2 My First Object

An object is a variable which type is not the one of fundamental data types of Visual Basic. To create an object whose type is Employee – the previous statement may also be phrased as “To create an object based on the Employee class”, we now move to the form. Remember, the form and its controls will be our client code. On the form do the following:

1. Add a button and a textbox control. Keep the generic names of the controls.

2. Change the Text property of the button to ‘Create Employee’. The button will be used to create a new object each time it is clicked. The new object will have the FirstName property set to whatever is typed in the textbox control.

3. Double click the button to add code to the click event handler of the button. Write the following code; the first line declares the object, and the second line sets the FirstName property.

   Dim Manager1 As New Employee
   Manager1.FirstName = Textbox1.Text

   Note the New keyword used here; this will be explained later.

With the code written, we can now test it to see how new objects are created, and their FirstName property set to whatever the textbox Text property contains. To monitor what’s happening to the first name, add a messagebox statement before and after the assignment statement above to display the value of Manager1.FirstName.

The only problem so far with the code above is that the object Manager1 is declared as a local variable under the Button’s click event handler. To access Manager1 from other event handlers, we need to move the declaration to the form level for a global scope of Manager1.
The declaration statement above does two things. First, it gives Manager1 its scope through the location where the statement is placed; later we will see how adding other keywords changes the scope. Secondly, the declaration statement of the above also creates a new instance of the Employee, through the use of the `New` keyword.

Later we will learn more about instantiation and the `New` keyword. For now, we need to know that objects, unlike Integer variables for example, do not exist in RAM without instantiation. Hence we need the `New` keyword to actually create Manager1. In other words, before using `New`, we do not have Manager1 in RAM. But, can we separate the scope from the instantiation?

If we want the Create Employee button to only create the Manager1 object, while we increase the scope of Manager1, we need the following changes:

1. Move the declaration to the form level and remove the `New` keyword:

   ```vbnet
   Dim Manager1 As Employee
   ```

   Note we removed the `New` keyword.

2. Under the Button’s click event handler add the following code:

   ```vbnet
   Manager1 = New Employee
   ```

   Note that the line above uses an assignment and the `New` keyword. This kind of a statement is called an instantiation. The declaration statement of the previous step only gives Manager1 its scope. By itself the declaration statement of the previous step does not create Manager1 in RAM. The instantiation statement of this step actually creates the object in RAM. Besides the declaration statement, the instantiation statement is the only other statement that may use the `New` keyword in the client code.
Now Manager1 is available for any event handler on the form due to the declaration statement being at the form level. But why did we need to move the `New` keyword, and what's the purpose of the assignment statement we typed at line 5 as shown above?

By moving `New` to under the Button’s click handler, we are now able to get a new object in RAM each time the button is clicked. If the New keyword was kept in the declaration, then only one object is being created during the lifetime of the form, and that creation happens during the form loading once the start button is clicked.

You may keep the `New` keyword in the form-level declaration and in the instantiation, but then the code is not consistent in when a new Employee is generated. This may lead to logical errors later, and should be avoided.

You can now go back to the Employee class code and add the property code for the `LastName` property. Remember to add the needed controls on the form - the client side- to set and manipulate the property. Below is the code for the DOB property.

```vbnet
Private Dim mDOB As Date
Property DOB() As Date
    Set (ByVal Value As Date)
    If DOB < Now() Then
        mDOB = Value
    End If
    End Set
Get
    Return mDOB
End Get
End Property
```

Note that the Set code is executed when the client code (e.g. the code on the form) uses:

```vbnet
ObjectName.PropertyName = SomeValue
```
In that case, SomeValue is passed into the Value variable shown above. Appropriately, we chose Date as the data type for the property in this case.

Now move to the form, and do the following:

1. Add a Textbox control and keep its generic name.
2. Add code under the button's click event handler, as shown in the following:

   Manager1 = New Employee
   Manager1.FirstName = Textbox1.Text
   Manager1.LastName = Textbox2.Text
   Manager1.DOB = CDate(Textbox3.Text)
   'to check that the above code has executed properly add this line:
   MessageBox.Show("New Employee created is ", & Manager1.FirstName & _
   " ", & Manager1.LastName & " with Date of Birth ", & Manager1.DOB.ToString)

The importance of the code above is that it first uses the FirstName, LastName and DOB properties’ Set code in the Employee class, when executing lines 2, 3 and 4 from above. Later the MessageBox statement calls the Get codes of the properties in the Employee class. To see this for yourself, add MessageBox statements as shown below, but remove them later as those are added only for you to understand how the Property code works.

   Private Dim mDOB As Date
   'in the above code, the variable is declared with the Private keyword
   'we will learn what that means and why later.

   Property DOB() As Date
      Set (ByVal Value As Date)
         If DOB < Now() Then
            MessageBox.Show("You are attempting to set DOB")
            mDOB = Value
            MessageBox.Show("DOB has been set")
         End If
      End Set
      Get
         MessageBox.Show("You are requesting the value of DOB")
         Return mDOB 'we do not place code after this line
         'Return above causes any code placed here not to execute
      End Get
   End Property

Moving on, add the code to define the Salary property as shown below. This code is part of the Employee class.

   Private mSalary As Single 'note the type chosen
   Public Property Salary As Single
      Get
         Return mSalary
      End Get
      Set (ByVal Value As Single)
         If Value > 0 Then
            mSalary = Value
         End If
Else
    MessageBox.Show("Salary must be more than 0")
End If
End Set
End Property

Note in the above code, the Salary property is defined with a Single data type. This is because we will need to manipulate the salary amount as a floating point numeric value. Also note that we added the test Value > 0 to ensure that when a new salary is to be assigned, then it can't be 0 or negative. Other tests may be applied based on the logic.

Now add code to set and get the Salary property for Manager1 on the form side as shown below.

1. Add a Textbox control and keep its generic name.
2. Add the following code to the existing code of the click event handler of the 'Create Employee' button:

   Manager1.Salary = CSng(Textbox4.Text)
   'and to test the Get code of the property use
   MessageBox.Show("The new salary is " & CStr(Manager1.Salary))

Test the code above by typing negative numbers in the textbox and you will see that those illegal values will not be assigned as the new salary.

**Important Note:** We are violating one of the main design characteristics we mentioned before in section 2.1, and that is the class code should not make any assumptions about the client code. We have already assumed that the client code is a Windows application, and that is not recommended. Do you recognize which statement in the class code makes that assumption? It's the MessageBox statement, since it is only available for Windows applications. At this point, we will accept this violation since we are using these messages to help us understand how the code works. Later we will learn alternatives to issuing those error messages without having the class code use MessageBox statements.

### 2.3 Going Public or Staying Private

The variables declared with the prefix m in the class are preceded with the `Private` keyword. This is done for a good reason. To see why it is important to use the `Private` keyword, let us try the `Public` keyword instead. Use the following code:

```vbnet
Public mSalary As Single
```

Run the project, and try to assign the salary a value by typing a positive number in the salary textbox, then click the button.

Now, type a negative number in Textbox4 and click the button, and you will get an error message stating that "Salary must be more than 0" and the salary stays at the old positive value. All of these results are as expected.

Since the mSalary variable in the Employee class was declared Public, then that means that variable is accessible from anywhere and any file in the project. In this case we can change the code under the 'Create Employee' button, so that it assigns mSalary to the value in the textbox, instead of using the Property Salary. So we will change the code under the click event handler of the button from:
Manager1.Salary = CSng(Textbox4.Text)
've and to test the Get code of the property use
Messagebox.show("The new salary is " & CStr(Manager1.Salary))

To

Manager1.mSalary = CSng(Textbox4.Text)
'above code does not call Set code-below code does not call Get code
Messagebox.show("The new salary is " & CStr(Manager1.mSalary))

Now run the project and type a negative number in the textbox. You will notice the new illegal value is accepted and the messagebox now shows the new salary at the negative number. As you can see, making those member variables public has two disadvantages:

1. Making a member variable Public allows the form code (client) direct access. The form code is free to assign any illegal value and therefore erasing previously legally stored values. Keep in mind that the form programmer may not be you! The class designer has no control over what the programmer of the client, unless the class members that are meant to be hidden or encapsulated- are declared private.

2. The Property Set code is bypassed through the direct access to its member variable, and therefore the logical checks performed to ensure a legal value assignment are now ignored.

Hence, to ensure that the Employee class only stores legal information and maintains its integrity, keep Private all the member variables that the property definitions use to store their data, and use Public access only for the property definitions. This enforces another OOP principle, which is the encapsulation principle; keeping the data itself hidden from the end user, and only allowing access through the property code.

Make sure you go back to the Employee class code and change the mSalary to Private, and change the code under the form's button to use the property name instead. Note that once you change mSalary to Private but still try to change it in the form code, you will get a compile error.

Incidentally, the member variable mSalary uses the prefix m in its name, which is short for member. This is only a naming convention.

2.4 My First Method

Assuming that on the end-user side, we need to give someone a raise of 7%. Without any changes to the Employee class, the end user code must calculate what the new salary should be and then assign it to the Salary property. That means the end-user programmer must do the work, and it is up to them to decide the rules governing the raises. However, to make the Employee class logically correct, we must write the code inside the Employee class that governs how the raises are given.

In that case we need to write a method (a method is a Function or a Sub procedure) that applies those raises, and changes the Salary to the new value. You may be thinking that the end-user programmer can assign that Salary to any positive value they want without the Employee class control; we will get to that imperfection later.

A method in a class code may be a Function or a Sub procedure. It is used to allow objects of that class to "do" things. In our example, we need the Promote method to calculate what the new salary should be based on a raise of 7%. Here is the code:
Sub Promote() 'a method may be defined to have arguments
    Salary = Salary * 1.07 'this line calls Get code then Set code
End Sub

The code above assigns a new value to the Salary property. Note that in the assignment statement above, the right side of the = sign activates the Salary property’s Get code to return the current value of the salary. That value will be multiplied by 1.07, and the result will be stored in the Salary property, hence the left-hand side of the assignment statement above activates the Set code of the Salary property. Recall the Set code of the Salary property checks for the legality of the new value before it is assigned to the mSalary variable.

Now let’s try it on the client code. Add the following code to the form code view:

1. Add a button control, and keep its generic name of button2.
2. Add code for button2’s Click event handler as shown below:
   
   Manager1.Promote()
   'and to check what the new Salary is:
   MessageBox.Show("The new Salary is ", & Manager1.Salary)

Test the code and use a calculator to ensure that the new salary is indeed a 7% raise from the old salary.

At this point, the Promote method does not affect the current rank or days in this position as we intended the logic to be. To make the Promote method comply with the logic, let’s complete the class code and define all the other properties of the class. The code below is the Employee class definition, phase 1.

Class Employee

    Dim mFirstName As String
    Dim mLastName As String
    Dim mDOB As Date
    Dim mSSN As String 'note not a numeric type
    Dim mDepartment As String
    Dim mSalary As Single
    Dim mDOH As Date
    Dim mDCP As Date
    Dim mRank As Integer 'ranges from 1 - 10

    Public Property Salary As Single
        Get
            Return mSalary
        End Get
        Set (ByVal Value As Single)
        If Value > 0 Then
            mSalary = Value
        Else
            MessageBox.Show("Salary must be more than 0")
        End If
    End Set
End Property

    Public Property FirstName As String
        Get
            Return mFirstName
        End Get
        Set (ByVal Value As String)
            mFirstName = Value
        End Set
End Property

14
Return mFirstName
End Get
Set (ByVal Value As String)
    If Value <> Nothing Then
        mFirstName = Value
    End If
End Set
End Property
Public Property LastName As String
    Get
        Return mLastName
    End Get
    Set (ByVal Value As String)
        If Value <> Nothing Then
            mLastName = Value
        End If
    End Set
End Property
Public Property SSN As String
    Get
        Return mSSN
    End Get
    Set (ByVal Value As String)
        If Value <> Nothing Then
            If Value.Length = 9 And IsNumeric(Value) Then
                'extra checks may be added
                mSSN = Value
            End If
        End If
    End Set
End Property
Public Property Department As String
    Get
        Return mDepartment
    End Get
    Set (ByVal Value As String)
        If Value <> Nothing Then
            mDepartment = Value
        End If
    End Set
End Property
Public Property DOB() As Date
    Set (ByVal Value As Date)
        If Value < Now() Then
            mDOB = Value
        End If
    End Set
    Get
        Return mDOB
    End Get
End Property
Public Property DOH() As Date
    Set (ByVal Value As Date)
Chapter 2

VB Classes and Objects

If Value < Now() Then
    mDOH = Value
End If
End Set
Get
Return mDOH
End Get
End Property

Public Property DCP() As Date
' DCP is Date in Current Department
Set (ByVal Value As Date)
    If Value < Now() Then
        mDCP = Value
    End If
End Set
Get
Return mDCP
End Get
End Property

Public Property Rank As Integer
Get
    Return mRank
End Get
Set (ByVal Value As Integer)
    If Value <= 10 And Value > 0 Then
        mRank = Value
    End If
End Set
End Property

Public Sub Promote()
' note the Public keyword so client can all this method
    Salary = Salary * 1.07 ' Cast 1.07 to Double
End Sub

End Class

Once you have completed the code for the Employee class, update the form code so that it uses more textboxes to allow the user to enter values for each property, and have those values assigned to the corresponding Manager1 properties under button1.

To make the code more interesting, make all the textboxes invisible except for the Salary textbox once the Manager1 has been created and its initial properties set.

However, there is a bug in the code of the Employee class. The end-user programmer can bypass the Promote method, and assign any salary value, making the employee class logic invalid. To block the end-user programmer from changing the Salary value, other than during the creation of the object Manager1, we must change the property definition of Salary, so that the Set code is deleted. This causes the property to only function when the end-user code is trying to evaluate the salary, for example to show it, but not assigning it to any value. There is a formal way of blocking the Set code by making the property aReadOnly property.

2.5 ReadOnly and WriteOnly Properties
The need for `ReadOnly` and `WriteOnly` properties may stem from many logical reasons, ranging from hiding secure information like social security numbers and passwords, to ensuring that the property value is safely controlled within the logical bounds of the reality, as in a salary range, raises, and promotions.

To define a `ReadOnly` property, remove the Set code, and add the `ReadOnly` keyword. The following shows an example.

```vbnet
Public ReadOnly Property Salary() As Single
    Get 'checks may also be added under Get for security
        Return mSalary
    End Get
    'Set code must be removed
End Property
```

However, once you do the above, no code can assign the property Salary to any value; not even inside the Employee class itself. The solution is to set the value during creation-time under `Sub New`, the constructor. We will learn about constructors in the next section, but first, we must change the Promote code because it uses the Salary property in a way that calls the Set code, and that Set code is not available. Hence we must change the Promote method to the following:

```vbnet
Public Sub Promote()
    mSalary = Salary * 1.07
    'Salary is ok on the Right side as that activates the Get code
End Sub
```

Now if you run the project, you will notice that you are unable to assign an initial salary to a new employee, since nowhere can you assign a value to the Salary property. The following snapshot shows what happens on the client side:

![Windows Application Form](image-url)
Hence, when a property is made **ReadOnly**, the client code may not assign it a value. However, undesirably, the class code may not assign a value to a **ReadOnly** property either. The alternative is to access the property's hidden variable instead from within the class code. But what about **WriteOnly** properties?

To define a **WriteOnly** property, we add the **WriteOnly** keyword in the property declaration, and remove the Get code as the following code shows:

```vbnet
Public WriteOnly Property Password() As String
    Set(ByVal value As String) 'checks may be added before assignment
        mPassword = value
    End Set
    'Get code must be removed
End Property
```

Just as **ReadOnly** prohibits the assignment of the property from anywhere, the **WriteOnly** prohibits the evaluation of a property from anywhere. Hence the following statements cause syntax errors:

```vbnet
Messagebox.show(Manager1.Password)  'syntax error in client code
UpdatedPassword = Password & “123”  'syntax error in class code
```

The solution is to not to evaluate the property in the client code, and use the hidden member, mPassword, in the class code.

### 2.6 Default Properties

Many experts argue that no code should use default properties. It is perhaps true, that code that uses default properties is confusing and hard to maintain. So you should refrain from defining or using default properties, but you should learn about them since they are part of OOP in Visual Basic.

As a programmer of VB in Windows applications, you have probably been aware that most of the controls have default properties. For example, a Textbox control has the Text property as its default property. This means that you can assign that Text property a value in code without specifying the property name. Hence the following two lines of code are equivalent, but the first one is preferred:

```vbnet
Textbox1.Text = “Some Value”  'is the same as
Textbox1 = “Some Value”
```

Formally, any property that accepts arguments can be declared as the default property for a class. A default property is a property that is assumed when no specific property has been named for an object.

**Caution:** Default properties are useful because they allow you to make your source code more compact by omitting frequently used property names. However, using them is not a good programming practice since most other languages do not use Default properties. Their use may make the code look unclear and harder to maintain.

The best candidates for default properties are those properties that accept parameters and that you think will be used the most often. For example, the Item property is a good choice for the
default property of a Listbox class because it is used frequently. The following rules apply to
default properties:

- A default property may not be **Shared** or **Private**. (We'll cover Shared later)
- Default properties must accept at least one argument. An example is shown below.

**Example:** The following example declares a property that contains a String as the default
property for a class. Just focus on the use of the **Default** keyword, the argument list in the Class
code, and the omission of the property name in the client code.

```vbnet
Class Class2
    Private PropertyValue As String

    Default Public Property Prop(ByVal a As Integer) As String
        'Note the use of the Default Keyword
        'The property must accept an argument as shown after the property name

        Get
            Return PropertyValue
        End Get
        Set(ByVal Value As String)
            PropertyValue = Value
        End Set
    End Property
End Class
```

2.6.1 Accessing Default Properties

You can access default properties using abbreviated syntax. For example, the following code
fragment uses both standard and default property syntax:

```vbnet
Dim C As New Class2()
' The first two lines of code access a property the standard way.
C.Prop1(0) = "Value One"    ' Property assignment.
MessageBox.Show(C.Prop1(0)) ' Property retrieval.

' The following two lines of code use default property syntax.
'Note property name is not used but the index must be.
C(0) = "Value Two"         ' Property assignment.
MessageBox.Show(C(0))      ' Property retrieval.
```

Confusing? It probably is, which is why this format is not recommended. Note that you don't have to declare a Default property, but it's an option that is available in VB. Also note that you can only assign one property to be the default property per class.

In the next chapter, we will learn more about the **New** keyword and work more with **ReadOnly**
and other properties.

**Exercises**

2.1 Write the definition of a class that represents a Car. Include the definition of these
properties: VIN Number, GasLevel, Mileage, and MPG (Miles Per Gallon). Make sure
to choose proper data types based on how these properties are used. For example, VIN Number, even though a number, but does not need to be defined as Integer.

2.2 Having done the code for the above question, write the code for a Drive method, which accepts a value that represents the number of miles driven. Drive will update the Mileage property, and reduce the GasLevel based on the MPG and the miles driven. This method should allow the car to be driven based on whether there is enough gas in the tank. If the miles passed require more gas than what is in the tank, drive the car till the tank is empty, and adjust the mileage only by what the actual miles driven were.

2.3 Write another method, PumpGas, which accepts a value that represents the number of gallons pumped. This method updates the GasLevel by adding the value of the argument received to the current GasLevel. What should the data type of the argument be?

2.4 Now design a form and write code for the buttons to create a new car, drive it, and pump gas into it. Values for miles driven and gas gallons may be obtained from the end user via textboxes or Inputboxes. Below is a suggested GUI for the form.
Chapter 3 Constructors

When the client code uses the *New* keyword in the instantiation statement, that causes the compiler to call a sub procedure named Sub New. If a class does not define that procedure, the compiler creates a built-in one. This procedure call and this procedure are important because now you know that you can write your own Sub New, so it accepts arguments. Those arguments may be values that you want to pass to the class code so that they are stored in certain variables—hidden member variables, especially those that hold values for *ReadOnly* properties.

Recall that a *ReadOnly* property causes the client code not to be able to assign the property any values. Using the *New* call, the client code is able to pass values to the class code and those values may be assigned to the properties’ private member variables. This chapter provides more details on Sub New and its uses.

### 3.1 Constructor method

Here is what we will do for the Employee class. We will create Sub New, and make it accept one argument. That argument’s value will be stored in the mSalary variable. The code follows:

```vbscript
Class Employee
    Dim mFirstName As String
    Dim mLastName As String
    Dim mDOB As Date
    Dim mSSN As String 'note not a numeric type
    Dim mDepartment As String
    Dim mSalary As Single
    Dim mDOH As Date
    Dim mDCP As Date
    Dim mRank As Integer 'ranges from 1 - 10

    Sub New(ByVal InitialSalary As Single)
        If InitialSalary > 0 Then
            mSalary = InitialSalary
        Else
            MessageBox.Show("illegal value for initial salary")
            'assign some default value otherwise initial Salary is 0
            mSalary = 20000.00
        End If
    End Sub

    Public ReadOnly Property Salary() As Single
        Get
            Return mSalary
        End Get
    End Property

    ....complete the rest of the class code
End Class
```

Now on the form side, under button1, we will change the code from:
Manager1 = New Employee
To
Manager1 = New Employee(Csng(Textbox1.Text))

So that when the line above executes, it will call Sub New with a Single type argument, and this Sub New has been defined already in the Employee class.

So does that mean that Sub New in the Employee class is a sub procedure just like any other procedures? The answer is yes, but not like all other procedures. It has a special status because of its name. The compiler recognizes that New is a keyword, and when it is placed in the form code as we wrote it, then that constitutes a call to Sub New in the Employee class, with the argument enclosed in the parenthesis after the Employee keyword passed to the sub procedure. Sub New is what we refer to as the class constructor.

If you are thinking, but I never typed a constructor before, how did it work? If you do not define a constructor in your class code, the compiler creates one for you (an argument-less one) by default. This allows the client code to 'call' this default constructor through the use of the New keyword in a declaration or instantiation statement. However, when you write your own constructor the compiler will no longer add the default one for you. So if you would like the form code to use the argument-less call, you should write an argument-less Sub New in addition to any other version of Sub New. We will provide examples later in this chapter.

You may, if you wish to give the form code more flexibility, write a number of constructors in the class, allowing the form code to call New with a minimum of the salary amount, or in addition pass along the first name, last name, and DOB.
To do so, just use the following code:

```
Sub New(InitialSalary As Single, FN As String, LN As String, DateOB as Date)
    'You may add code to validate the values passed above
    mSalary = InitialSalary
    mFirstName = FN
    mLastName = LN
    mDOB = DateOB
End Sub
```

If you write the code above in addition to the Sub New we wrote before:

```
Sub New(InitialSalary As Single)
    If InitialSalary > 0 Then
        mSalary = InitialSalary
    Else
        MessageBox.Show("illegal value for initial salary")
    End If
End Sub
```

Then this allows the form code to use any one of:

```
Manager1 = New Employee(Csng(T1.Text),T2.Text, T3.Text, CDate(T5.Text))
Or
Manager1 = New Employee(Csng(T1.Text))
```
And later code would assign the values of the first name, last name and date of birth. The compiler 'knows' which version of Sub New to call in the class code based on the number and type of arguments passed in the calling code.

3.2 ReadOnly and Sub New

As stated before, to define a ReadOnly property, use the ReadOnly keyword and do not write the Set code. Here is the example for the Salary property:

```vbnet
Public ReadOnly Property Salary() As Single
    Get
        Return mSalary
    End Get
    'Set code all gone
End Property
```

This new change means that if the end-use code uses a statement like:

```vbnet
Manager1.Salary = CSng(TextboxName.Text)
```

They will get a syntax error stating the property is ReadOnly and may not be assigned, which is the required effect of the Read Only property. So in general, you need an argument in the constructor for every ReadOnly property value. The constructor code would assign a value passed from the end–user code during instantiation.

3.3 Tying it together - General Design Issues

It is sometimes easier to learn by writing code first then going back to the theory and examine how it was applied. This section emphasizes the application of the concepts learned so far, and how to apply them when designing your classes and objects.

Previously we learned that at the heart of OOP is a Class that defines an object. The important design issues to remember when creating a class are the following:

1. **Encapsulation**, or hiding details: The client code should not have access or 'see' the details of the class definition. This requires hiding (by declaring Private) some members of the class code. When an object is created, the user code has access only to Public properties and methods. Data is encapsulated as the values of the properties, how they are assigned, and how they are changed is not visible to the user. This can only happen if the data members holding the actual values of these properties are declared Private. In short, declare the data members Private; manipulate these members through the Property code and methods and allow the user to access values by declaring the properties and methods Public.

2. **Code re-use** through the creation of objects: Again, one of the powerful features of OOP, is through using an object. The programmer does not need to redefine the entire code (Class code) declaring that object. If the class is compiled, which is the case for most predefined classes, the programmer of the client code only needs to read documentation about the publicly available members (Properties and Methods). If the class source code is available, then proper analysis will reveal how the class works. For
example, a programmer, as a class user (client code), does not have access to the source code of a Textbox class. A Textbox class is one of the built-in classes, so the programmer working on a form is able to add instances of Textboxes to the form, and later assign and evaluate properties, and call methods of the Textbox class.

Having stated how the client, or form code, should not access the to-be-hidden data in the class code, the class code has also some design issues that need paying attention to. The following are guidelines to be followed when designing classes:

1. When designing a class, remember that in most cases, the user code, or the programmer of the user code, is NOT you. For example, you are a programmer who adds controls and forms to a project. You are not the same person who wrote the class code of these controls. Hence when designing a class, ‘wear’ the designer hat and hide the class data from direct manipulation by the user code.

2. Similarly, the class code should be independent from any user code, so that the class will work with any form and controls used. Based on this technique, the class code should use no form or control names. Further, the logic of the class properties and methods should not depend on the logic of the user code. For example, do not rely on the user code to check that the range of a certain property value is valid. Make sure that all range and data validation for the properties and methods is done inside the class code. To be more general, the class code should not make any assumptions about the client code type. For example, the client code may not be a Windows application.

3. There may be occasions when the class needs to communicate with the client code. For example, when the user-code violates a logical rule of a property assignment, the class needs to communicate this error. This communication is better kept private, so the end-user of the application does not ‘see’ this interaction. So far, we have been using a MessageBox statement from the class code, but this is not a private communication, and using a MessageBox assumes that the client code is a Windows Application which may not be the case. To make this communication private, we will learn how to use Events and Exceptions in the future to allow the class code to communicate with the end-user code; hence the communication will be between code elements not code and end-user. For now, view these MessageBox statements as placeholder that will be replaced with either Events raised, or Exceptions thrown in the future.

3.4 Private and Public Methods

In general we can assume that when declaring properties you want those to be available to the client code, hence you declare them Public. However, methods, which are the Sub procedures and Function procedures that a class uses, may not all declared Public. It depends on which methods are to be accessed by the client code and which ones are made for the sole use of the class itself.

If a method is only to be used by the class itself, then declare it as Private. Later when we learn about inheritance, you may also use the Protected keyword for a different access level. We will learn about the different access keywords once we cover inheritance.

For example, you may want to write a method that calculates the raise amounts for Employees based on a secret code assigned to each employee. This method should not be made visible to the client code, so declare it as Private. Then when the client code uses the Promote method, which is Public, the Promote method would call this ‘hidden’ method as shown below:
Public Sub Promote() 'note the Public keyword- we will cover it later
    'instead of mSalary = Salary * 1.07 , we will use the next line
    mSalary = SecretPromote()
End Sub

Private Function SecretPromote() As Single
    If Rank >= 5 Then
        Return (Rank * 1.09 * mSalary)
    Else
        Return (Rank * mSalary)
    End If
End Function

Given that SecretPromote is declared Private, the client code may not call it.

3.5 Is Nothing

When comparing values of type Integer, you use the equal sign as in the following example, where both variables x and y where declared Integer:

    If x = y Then

However, when comparing objects, the equal sign works differently and may produce unexpected results. We will cover this particular topic (assigning objects and comparing with equal) in a later chapter. For now, when creating an object, then from a logical point of view, comparing that object to a single value does not make sense, as the following code shows:

    Dim O As New Employee
    'code to assign O's properties
    If O = Q  Then  'however, what does Q represent?

The only comparison that makes logical sense when comparing objects is to check whether the object has been instantiated or not. This is done through using the Is operator and the Nothing keyword. This check is important because when you declare an object at form level, then later code may want to assign that object’s properties to values before it has been instantiated, which leads to a runtime error. One way to avoid such situation is to use the Is Nothing check.

When a new variable of a fundamental data type, such as Integer, is declared, it is initialized to its default value, 0 for Integer. So when an object is declared, what value does it default to? Further, if a programmer needs to reset a fundamental data type variable, the variable is assigned to its default value. But how can a programmer achieve the same effect for objects?

Object variables have no actual default value like fundamental data types do. However when an object is declared but not instantiated then that object may be compared to the keyword Nothing. This keyword represents no actual value which is why we don’t say it is a default value. It only indicates that the object has not been instantiated, and therefore may not be used yet – till New is called. Nothing in VB is equivalent to Null in other languages.

Now, since Nothing is not an actual value, then we may not compare an object to this keyword using a regular equal sign. Instead, we must use the special operator, Is, as in the following code:

    Dim O As Employee  'at the form level
'under some button handler we need to check if O has been instantiated:
If O Is Nothing Then
    MessageBox.Show("Enter values then hit Create Employee first")
Else
    O.Promote()
End If

3.6 Killing Objects

The keyword **Nothing** can be used to check if an object has been instantiated, and it can also be used to ‘de-instantiate’ an object. In other words, to reset an object, assign it to **Nothing**, as in the following code in which MyEmployee is of type Employee and has been instantiated already:

```
MyEmployee = Nothing
```

As stated in the previous section, **Nothing** is also what an object starts at when declared and not instantiated as in the following code:

```
Dim MyEmployee As Employee 'MyEmployee is Nothing at this time
```

It is worthwhile to mention here that when an object falls out of scope, then the compiler takes care of killing it and releasing its space in RAM. For example in the following code, object O is killed by the compiler by the End Sub statement.

```
Sub LiveAndLetDie()
    Dim O As New Employee
    O.FirstName = "John"
    'more code to manipulate O
End Sub 'by this line, O has been killed in RAM
```

Given the above code, it does not hurt if the code explicitly kills object O by assigning it to Nothing. It is a good programming practice, and makes the code easier to analyze.

**Exercises**

3.1 The Car class created in the previous chapter used the GasLevel and MPG properties. Those properties must be changed to ReadOnly. Update the property definitions, and add two constructors for the car class. One that accepts the initial GasLevel and MPG values and another that accepts those in addition to the Mileage.

3.2 Adjust the GUI to the car class above, so that you can now kill a car when needed.

3.3 Also adjust the form code so that before a car is driven or pumped with gas, code checks that a car ‘has been created’.

3.4 If you hit the Create car button, an object of type Car will be instantiated. If you hit the button one more time, another object of type Car will be created. What will happen to the first created car?
Chapter 4 Events

We have stated earlier that when designing a class, the class code should not refer to any of the client code, its control, nor should it assume that the client code is a Windows application or any type. In fact the client code may be a Web ASP application or even be written in C# or C++ .NET languages.

In addition to not making any assumptions about the client code, the class code should not communicate with the end-user either. This follows from the fact that the class code should not make any assumptions about the client code, because without knowing what type of client code is used, the class code cannot include specific statements like MessageBox.show to communicate any messages or warnings. MessageBox.show statements are only available to Windows types of applications.

A class may need to communicate with the client code on some information, especially if there is a violation of the data logic, or if a fatal error situation may arise. In order not to communicate publicly so the end-user witnesses that communication, the class needs to communicate in private and without making any assumptions about the nature of the client code. For example, if the client code attempts to assign an illegal value to a property, then if the class code issues messages via MessageBox.Show which are displayed directly to the screen (end-user), then that communication is public and assumes that the client code is a Windows application. In order design the class code to communicate in private through code only, the class uses an Event instead.

Before we go into details about events, we need to stress that there are two types of communications that a class may need to perform:

1. Communications regarding a warning or to convey a situation that is not fatal. This type of communication must be done through events. Examples are when a value is legal but is lower than a certain threshold, when a button is clicked, when a textbox content in changed, and so on. You may be familiar with some of these concepts since as a programmer you have been writing code when some of these events have occurred, for example, when a button is clicked.

2. The other type of communication occurs in more serious cases that require immediate action. For example, when the user assigns an illegal value to a property, when the salary assigned is negative, or above a million, or when the department code is not allowed. These cases warrant an error or runtime exceptions. We will cover Exceptions, and Exception Handling in a later chapter. Until then, we will still use MessageBox statements to display error messages on the screen as a temporary solution.

Note In all of the communications the class will perform, we are “sending” messages to the client code not the client or end-user.

In this chapter we will learn topics related to events, such as event declaration, event firing, and event handling.

When dealing with events, there are two major codes: Code that fires or generates the event, called event source, and code that responds or handles the event, called event sink. The event source is typically a class code. Firing the event is meant to send a message to the client code. In order for a class code to fire events, there are two code elements needed for each event: Code
to declare the event, and code to generate the event. While there must be only one statement to declare the event, there may be many statements throughout the class code that generate or fire the event, based on the logic.

If you want to draw parallels to Click event handlers, just remember that in writing the Click event handler of a Button, you are writing the handling code. The code that raises that event is not visible to you as a programmer using the Button object.

4.1 Event Declaration

An event is much like a variable or a procedure, in that it must be declared. To declare an event, you need to choose an event name, and use the `Event` identifier as in the following statement:

```vbnet
Event LowBalance()
```

And here is another one:

```vbnet
Event SalaryAboveMillion()
```

Unlike a variable, and more like a procedure, an event usually uses arguments to pass on some information to the recipient. So if an event is going to pass information in an argument list, that list must be declared. Here are some examples:

```vbnet
Event LowBalance(threshold As Single) 'ByVal is the default-omitted here
```

And another one:

```vbnet
Event SalaryAboveMillion(ByVal RankCode As Integer)
```

4.2 Event Firing

Once an event has been declared, the class code may ‘send’ it. The process of sending an event also referred to as firing the event is done through the `RaiseEvent` statement.

Here is an example of the `RaiseEvent` statement for the declared event `SalaryAboveMillion` shown above:

```vbnet
RaiseEvent SalaryAboveMillion(4)
```

Normally an event is raised when certain logical conditions are met, and those conditions are decided by the logic of the class. For example, in the Employee example, if the salary of some manager in certain ranks is increased several times and at some point it will be exceeding a million dollars, then the class should issue a warning, so that the end-user working the payroll system is aware and does something about it. To start this process, you need to declare the `SalaryAboveMillion` event and then raise it in the Employee class code. Based on the logic, we may wish to raise this event whenever the salary amount is being changed. This includes the code under the Promote method, and if the Salary property is Read-Write, then also under the Salary property’s Set code. Here is an example:

```vbnet
Public Sub Promote()
    mSalary = Salary * 1.07
    mSalary = SecretPromote()
    If mSalary >= 100,000,000.00 Then
        RaiseEvent SalaryAboveMillion(Me.Rank) 'Me.Rank is current rank
    End If
End Sub
```
So now that you know that an event is nothing but a message being sent, it is important to know whom the recipient is.

When a class raises an event, called event firing, the event raised is like a Sub procedure call made to another Sub procedure. Because the intent is to communicate a message between the class and the client code, then we initiate the call in the class code, while the call is to a Sub procedure in the client code. However the called procedure has not been written yet, so the class cannot use a Call statement. Instead, the raised event is like a call to a procedure that may be written to respond to or handle the raised event. This special procedure that may be written is called the event handler, also referred to as event sink. The client code must contain certain statements or clauses that connect the event handler to the event source, as we shall learn later in this chapter.

**Note** that an event may be left unhandled, and that situation is not unfamiliar. For example, a textbox control has tens of events, but you may create a completed Windows application with many textboxes, none of which has any event handlers.

You should be familiar with event handlers, since you have been programming Windows applications that use forms and controls. For example, the Click event handler, which is a very common event handler for a Button, is code that is called when the end-user clicks the corresponding button during the runtime. When such an action occurs, the compiler calls—the call is generated through the Click event being raised in the Button class code— the event handling code if it is written in the form’s code file.

It is important to note that an event must be raised within the scope where it is declared. For example, if you create a class XClass, and define an event YEvent, then only that class, XClass can raise that event, meaning the RaiseEvent statement can only be used within the class definition. So a form’s code may not raise that event. When we learn about inheritance in later chapters, we will learn more about event-raising features as well.

Now that the event has been declared and raised, we decide what to do in the client code.

### 4.3 Connecting Events with Event Handlers

You are probably familiar with event handling through writing event handlers for many controls in a Windows Forms application. The typical way to handle events for Windows controls is to double click the control in Form View, and that creates the event handler “code place holder”. You would then write the needed statements that must execute when that event is fired during the runtime. With these event handlers, the first line that defines the event handler ends with the “Handles ControlName.EventName” clause. This type of event handling is called Static Event Handling, and is not the only way to handle events.

Note that an event handler can only be a Sub procedure, as you cannot use a Function as an Event Handler, because it cannot return a value to the event source.

There are two methods to handle events: Static, which is the most common, and Dynamic, which requires more code and a deeper understanding of the logic. We will first cover Static event handling and then detail how to work with Dynamic event handling. Remember that it’s the handling that can be Static or Dynamic; raising events is not affected by how they are being handled.
It is important to mention here that while Static event handling is the standard way, Dynamic event handling, which requires a little more coding, gives the programmer more control over RAM usage and flexibility in code writing.

4.3.1 Static Event Handling

As a VB programmer writing code for Windows Applications you must be already familiar with Static Event Handling (SEH), but probably you were not aware of the name.

When you write code for an event handler for a control in a Windows application, that code is a SEH. For example, if you add a Button control to a form, and then write code so that when the user clicks on the button, that code would run, you would write the Click event handler for that button. You do that by double clicking the button in the form’s Design View, and then simply write the statements in between the event handler’s Sub procedure signature line and the End Sub line. What you did not do though, is declare that button, so you are not aware of what is involved in SEH during the declaration of the object that is going to handle events statically. When you add controls and then write event handlers for them, the VB IDE does a lot of work in the background based on the OOP principles we discussed before, so some of the details are hidden. In this section, we will detail the process involved in SEH.

Event handling in general requires two components:

1. A way to connect the event declared and later raised in the class code to the Sub procedure to contain the handling code.

2. The handling code statements.

Static and Dynamic event handling differ in the two components above. This section will detail how Static handling works based on those two components.

In theory, to handle an event statically, we need the following on the form (client) code:

1. An object that is declared using the ** WithEvents** keyword as shown below. This requirement only signifies that the object will use Static handling.

   ```vbscript
   Dim WithEvents MyObject as TypeName()
   ```

2. A Sub procedure must be written that has the same signature as the event. For this you need to view the documentation of the class if you don't have access to the source code, otherwise look up how the event was declared. This Sub procedure satisfies the second component that EH requires. In static event handling the following clause must be added to the right of the argument list of the handling code as shown below to satisfy the first component that EH requires; namely to link the handler to the event raised.

   ```vbscript
   Sub SubProcName(argumentList) Handles ObjectName.EventName
   'code must be written for the action in this case
   End Sub
   ```

**Important Note:** The code above should follow the same signature that the event was declared with in the class code. The code must be a Sub procedure and not a Function. The procedure name need not match the event name; only the argument list must match.
By default, when you write code in a Windows application, every control you add on the form is an object. So, for example, when you add a Button to the form, say button1, the IDE adds the following code:

```vbnet
Dim WithEvents Button1 As Button
```

Where, Button1 is the name given to the control object variable (object for short) representing the button you added on the form, and Button is the type name. Then if you want to write the code for the Click event handler, you will notice the Handles clause added to the Sub procedure to do the handling.

```vbnet
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click 'you add code here
End Sub
```

Under the Button class, somewhere there must code that declares the event:

```vbnet
Event Click(ByVal a As System.Object, ByVal b As System.EventArgs)
```

And somewhere else in the class, there must be code that looks like:

```vbnet
RaiseEvent Click(x,y)
```

Where x and y are already declared and assigned values.

The `WithEvents` keyword tells the compiler that this object will respond to events later. Without it, the compiler will not be able to 'link' the code you write (handler code) with the Button1 object Click event, when that event is raised.

**Recall** that an event raised is like a call statement, and the event handler is the called procedure. In SEH, the `WithEvents` keyword together with the `Handles` clause, will link the event being raised –when it is raised- to the event handler –if that handler code is written.

Visual Basic uses a standard naming convention for event handlers that combines the name of the object, an underscore, and the name of the event. For example, the Click event of a button named Button1 would be named Sub Button1_Click but this is only a naming convention. You may use this naming convention when defining event handlers for your own events, but it is not required; you can use any valid Sub procedure name.

At this point we will not dive into the details of how SEH works for controls, as we will cover how forms and controls function as OOP objects and classes in later chapters.

To practice this topic, we will build the code example of the Employee class. In the code the follows we are showing a portion of the Employee class that is related to the Salary property. If you have not developed the Employee class as per chapters 1 through 3, you should complete that class before moving on. Next we will add code to raise the event from the class, and then show how an object of type Employee on the form will respond to that event using SEH.
1. In the Employee class, add the following code to declare the event then raise it in the Promote method:

   Event SalaryAboveMillion(ByVal y As Integer) 'declare the event
   Public Sub Promote()
       mSalary = Salary * 1.07
       If mSalary >= 1,000,000.00 Then
           RaiseEvent SalaryaboveMillion(Me.Rank) 'raise event
       End If
   End Sub

2. In the form Code View, change the line that declares Manager1 to:

   Dim WithEvents Manager1 As Employee 'add the WithEvents keyword

3. The code for Button1 should not change; hence you should have the following code under Button1. Button1 is the Create Employee button.

   Manager1 = New Employee(999,000.00) 'startup salary
   'assign all other properties to textboxes – next is one example
   Manager1.FirstName = Textbox1.Text
   'complete assignment of all other properties

4. Make sure the Promote button code is written as:

   Manager1.Promote()

5. Next write the event handler for the SalaryAboveMillion Event:

   Sub SalaryAlert(a As Integer) Handles Manager1.SalaryAboveMillion
       'Note the Handles clause listing the objectName.EventName
       'Also note the argument list matches the class declared event’s list
       MessageBox.Show(Manager1.FirstName & " is earning above limits")
       'other statements may be added based on the logic
   End Sub

6. Run the project to test it and make sure to ‘promote’ the manager object several times so the salary triggers the event.

4.3.2 The Handles Clause and Static Handling:

The WithEvents keyword allows you to create object variables that can handle event statically. The event handlers in that case must use the Handles clause at the end of the sub procedure definition or signature line. In the previous section we walked through an example of how SEH works. In this section we will provide more details.

When declaring an object that will handle events statically, the object name in the declaration statement must be preceded with the WithEvents keyword.

   Dim WithEvents Accountant As Employee
**Note** that *Dim* becomes Optional when used with the *WithEvents* keyword. Hence the line above may also be written as:

```
WithEvents Accountant As Employee
```

**Note** also that objects declared using *WithEvents* must be declared at least at the form or global level, and may not be declared at the local scope level.

We already stated that when an event is raised, that constitutes a call to the event handler if it is written in the client code. In SEH, the compiler needs the Handles clause to link the event handler to the event being raised. Hence, an event handler in SEH must use the *Handles* clause, in addition to having the handler’s Sub procedure argument list matching the declared event argument list.

```
Sub SubName(ByVal x As Integer) Handles Accountant.EventName
    'any code here
End Sub
```

**Note** that the `ObjectName.EventName` portion may be a list of items separate with commas as in the next example.

```
Sub SubName(ByVal x As Integer) Handles A.Event1, B.Event1, C.Event2
    'any code here
End Sub
```

Where A, B and C are objects of one type, while the handled event is Event1 –the same– for objects A and B, while object C is handling a different event –Event2– altogether. The only restriction is that Event1 and Event2 must both be declared with a matching signature of `SubName`.

You can try the following example, which handles the Click event for two different Buttons. You can also add a Textbox control to the code as shown later.

```
Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button2.Click, Button3.Click
    'any code can be added here
    'you can check which button caused the event
    'by checking the variable *sender*
    'Example: If sender.Name.ToString = “Button2” Then ...
    '**caution:** In string comparision “Button2” is not the same as “button2”
End Sub
```

The following example shows how you can handle different object types, in different events. The only restriction is the signatures must match.

```
Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button2.Click, TextBox2.TextChanged
    'this is an example
    'we will learn more about the *GetType* method in chapter 5
    If sender.GetType Is TextBox1.GetType Then
        'any code here
    End If
End Sub
```
With static event handling you can write as many handlers as you need based on the number of events that are declared in the class. All you need for other handlers is to write the Sub procedure handler followed by the \texttt{Handles} keyword as we did above. This sounds easy, so there must be a disadvantage.

When you declare an object using the \texttt{ WithEvents} keyword, the size of that object in RAM will include all the handlers you wrote. This means lots of KB if you have many handlers. If you think that this should be ok, think about cases when handlers may not be invoked. They are there because they \textit{may} be needed, nevertheless, the object carries excess luggage that may not be used, and hence all that space slows the code down. Is there a solution? Yes, in Dynamic event handling, the object is created without any handlers associated, and when they are needed, a statement may create or destroy that association. This is where programmer's efficiency skills are required.

The other method is called \textbf{Dynamic event handling} and is based on using the \texttt{ AddHandler} statement. Dynamic Event Handling also requires the declared objects \textbf{NOT} to be declared using the \texttt{ WithEvents} keyword, and that the handler code \textbf{NOT} to use the \texttt{ Handles} clause.

\textbf{4.4 Dynamic Event Handling Using AddHandler}

Visual Basic .NET provides a second way to handle events using the \texttt{ AddHandler} statement in what is called Dynamic Event Handling, DEH. DEH uses the \texttt{ AddHandler} and \texttt{ RemoveHandler} statements to allow code to start and stop event handling for a specific event during the runtime. Note that you can start and stop the \textit{handling}, not the raising of events. You can use either SEH, or DEH, but you cannot use mix them for the same object.

The \texttt{ AddHandler} statement is used to dynamically connect an event with an event handler procedure. With this method, you \textbf{cannot declare the object using the WithEvents} keyword. Then the \texttt{ RemoveHandler} statement is used to break this connection. For every \texttt{ AddHandler} statement, there should be a \texttt{ RemoveHandler} statement for proper logical outcome.

\textbf{4.4.1 Handling events using Add Handler}

To use DEH, declare an object variable of the class that is the source of the events you want to handle. Unlike a \texttt{ WithEvents} variable, this can be a local variable in a procedure; for example, below CI is an object of type Class1:

\begin{verbatim}
    Dim CI As New Class1()  'you may also declare only and instantiate later
\end{verbatim}

\textbf{Note} that the \texttt{ WithEvents} keyword is \textbf{NOT} allowed in DEH. Also note that objects declared without the \texttt{ WithEvents} keyword \textit{may} have local scope.

Now, to connect a particular event named \texttt{AnEvent} of Class1 with a Sub procedure named \texttt{EHandler}, that is meant to handle that event, use the \texttt{ AddHandler} statement to specify the object name and the name of the event, with the \texttt{ AddressOf} clause to provide the name of the event handler; for example:

\begin{verbatim}
    AddHandler CI.AnEvent, AddressOf EHandler
\end{verbatim}
Where CI is the object name, AnEvent is the event name, EHandler is the Sub procedure name that you want to run.

Any procedure can serve as an event handler as long as it supports the correct arguments for the event being handled.

4.4.2 Using RemoveHandler to Stop Handling Events

You can use the `RemoveHandler` statement to dynamically disconnect events from the event handler procedures. This frees up the link to the handler code and hence makes the object CI above use less RAM. In addition, there may be cases where the logic requires the disabling of event handling. With SEH this disconnection is not possible.

Using the `RemoveHandler` statement is similar to the `AddHandler` statement. You must specify the object name with the name of the event, and then add the `AddressOf` clause to provide the name of the event handler as shown below.

```
RemoveHandler CI.AnEvent, AddressOf EHandler
```

Where CI is the object name, AnEvent is the Event name, EHandler is the sub procedure connected previously in an `AddHandler` statement to handle AnEvent.

The advantage of DEH is that while you stop handling using `RemoveHandler` or before you link to the handler using the `AddHandler`, the object is using less RAM space, hence you are running more efficient code. Another advantage is that you can later, based on the logic, link the object to another handler using the `AddHandler` statement, so the event can be handled differently based on different situations. For example, for a car, the service done at 30K miles is different than the one at 40K miles. So if a car class raises a NeedService event then based on the mileage, the handling code may be different.

4.4.3 Adding Handlers without Removing Them

The following is a logical situation that needs attention when using DEH. In SEH, the `WithEvents` along with the `Handles` clause on the Sub procedure to handle the event create the association between the object, the event, and when to call the handler.

As you know, in DEH, we associate when the handler code should run using the `AddHandler` statement. Attention should be paid to using `RemoveHandler`, for every `AddHandler`. The following example shows why.

Every time an `AddHandler` statement runs, the compiler registers that when the event referenced, call it A, fires then the handler referenced, call it P, should be executed. If the `AddHandler` executes for the same event and its same handler say 5 times, then the event handler P, will execute 5 times, which may not be desired logically. For example, if you add the following as the handler for an event named SalaryAboveMillion for an object named Clerk:

```
Dim Clerk As New Employee
```

Then each time Clerk is given a raise under some button the following is executed:

```
AddHandler Clerk.SalaryAboveMillion AddressOf ShowWarning
Clerk.Promote()
```
Then say the above code to promote the Clerk is executed 5 times. Then each time, because of the **AddHandler** statement, a Sub procedure named **ShowWarning** is “coupled” with the possible event **SalaryAboveMillion**. If the Salary of Clerk does go above million, triggering the event, there are 5 Sub procedures waiting to run, causing the message of the code below to show 5 times, where the following is the code for **ShowWarning**:

```vbnet
Sub ShowWarning(x As Integer) 'note no Handles clause in DEH
    MessageBox.Show(Clerk.FirstName & " rank " & x & " earns > 1M")
End Sub
```

The problem is that the **AddHandler** statement adds a Sub procedure “waiting” to handle the event. Once the event is raised, ‘all’ procedures ‘marked’ as the handlers of the event are executed. The solution for this problem is to use a **RemoveHandler** statement for every **AddHandler** used, as in:

```vbnet
RemoveHandler Clerk.SalaryAboveMillion, AddressOf ShowWarning
```

After the call to promote. This way, if Promote does not cause the event to fire, the the next thing to execute is to remove the ‘waiting’ handling code.

It is important to note here that the **AddHandler** statement is placed **BEFORE any code that may cause the event to fire**. If the Add statement is placed **AFTER the Promote method is called**, then the event will still fire, but there won’t be any sub procedure ‘waiting’ to handle it, so the event will go unhandled. Recall that not every event must be handled, but the logic of the application will dictate which events to handle and under what circumstances.

### 4.5 Sequence of Events

When an event is raised, which code runs first? The event handler or the line that follows the **RaiseEvent** statement? In other words, what happens when the compiler sees a **RaiseEvent** statement? Does the line that follow that statement get executed before the event handler gets called? This section details the sequence of execution that the VB compiler follows.

When code (typically in a class) raises an event, then the client code handler if written will be executed before the line that follows the raise event line. Hence, if the Employee class has the following code:

```vbnet
...  
mSalary = 5000  'line 1  
RaiseEvent SomeEvent 'line 2  
mSalary = 9000  'line 3
```

Then when the handler runs, mSalary is at the 'old' amount of 5000 because line 2 above raises the event, and what happens next is **NOT** the execution of line 3, but the execution of the event handler of Somevent, if it was written. This ‘jumping’ action tells you to make sure you do assignments (and other requirements) before you raise events. Needless to say that sometimes the logic requires the event to be raised first, so pay attention to this detail.

At this point it is important to note the following two facts:

1. **When an event is raised the compiler moves control to the client code if the client code does contain an event handler. If the class code contains any code following the **RaiseEvent** statement, that code will be executed AFTER the event handler is executed.**
This may cause logical errors, which will be explained shortly. For example in the following code, the MessageBox statement may not fire when expected:

```vbnet
'in class code
RaiseEvent EventName 'assuming EventName is declared already
Messagebox.show("Line following Raise Event")
```

2. The client code is NOT required to respond to every or any event firing. This is similar to email, when a person sends an email, the recipient is not required to respond. From a programming viewpoint, you have been adding controls to a form, but you don’t write every handler of every event of every control. You only handle the events that you think the logic requires.

Now that we have covered many basic features of OOP, is it time to win the prize. In the next chapter we move to inheritance, which is at the heart of OOP.

**Exercises**

4.1 Using the Car code class of chapter 3 exercises, add code that declares and raises an event named LowGas. This event is raised when the gas level reaches 5 or lower.

4.2 On the form side, add code so that when the LowGas event is raised, the event is handled by asking the user to pump gas using static event handling.

4.3 To practice Dynamic event handling, implement the handling of the LowGas event using dynamic event handling. Ensure that you remove the handler properly.
Chapter 5 Inheritance

At the heart of OOP is the concept of code reuse, and inheritance is at the heart of code reuse. Without inheritance, you cannot create forms that have different controls and codes behind them for each application. Without inheritance, you will have to write the code to create the rectangular shape of the form and all the behavior that it follows and that (trust me) is a lot of work!

With inheritance, Programmers are able to 'reuse' the code that already defines what a form is, what events it can respond to, and what methods you can activate on it. Further, with inheritance you can, if you want to, change how a form looks, and give it more properties to make it more suitable for your application. Just as you can customize a form, you can do the same for textboxes and other controls. For example, you can create a textbox that can only accept numeric entries, and call it a NumericBox. You may think, I can write the code to create the shape and then define the properties and events to make it work like a textbox but only for numbers. True, but without inheritance, you will have to write lots of code and spend numerous hours debugging, while with inheritance, the amount of work is a fraction.

5.1 What is inheritance?

In real life when you inherit a relative, you get their money. Sounds good; but in OOP, Inheritance is about the 'genes' you get from your relatives, whether you like those genes or not. So all the properties and methods of a parent are passed on to the child. Unlike real life, in OOP inheritance, you can improve the genes that are inherited so the child works better that the parent.

From a design point of view, inheritance usually is planned, and is not done per need basis. So, if you are planning to create an application that involves graphics, you would typically plan ahead and start by creating a generic class that other classes will inherit from and reuse its code. Let's take an example like drawing a geometric shape. You define the shape to have a size, number of sides, line color, line thickness, and other properties. You may also define the methods to draw the shape on the screen, etc. Later you realize that a circle is a shape but it is in a 'class' of its own that is 'different' from a square, although they do have some similar properties and behaviors. So what do you do? Do you create two classes, one called Circle, and one called Rectangle? The answer is yes. But do you 'repeat' all the methods you coded for drawing a circle into the rectangle class? No, you would pick all the properties and actions that both circles and rectangles are similar in, and put these in a one generic class and say call it Shape. Then you create other classes for particular shapes and code the specificity of each shape. These other classes will be child classes that inherit the generic Shape class.

You start by creating a class Shape. Then you define in that class, that a shape has size, lines, etc. Now you can create two new classes, one called Circle, and another called Rectangle. These two classes will inherit the Shape class, so that you can "re-use" all the methods that the Shape class has. In addition, you can add some specialized methods that have to do with circles in the Circle class. You can also "modify" how some of the methods inherited from the Shape class behave in the Circle class. You do the same for the Rectangle class.

5.2 Terminology

The inherited class, called base or parent class, will pass on all of its 'possessions' to its child class, otherwise also known as the derived class. Unless the private access modifier is used with
a method, all the parent methods and properties will be available for use in the child class. For example, if a Manager class inherits the Employee class, then an object of type Manager will have the FirstName, LastName, Salary, Department, and all other Employee properties without having to rewrite those properties in the Manager class. The Manager class is called the derived class, and the Employee class is called the base class.

In addition, the derived class is not ‘stuck’ with the base class's methods. It may redefine some of those methods that may need to be changed. The Promote method for example, through a process referred to as **overriding** or **shadowing**, may give different promotion rates to objects of type Manager. Both properties and methods may be overridden or shadowed. This way an object of type Manager will get promotions based on different conditions than an object of type Employee.

### 5.3 Inheritance and "Is a" Relationships

In OOP, classes may be related in a different way, where a child class is typically a special case of the parent class. This relation is called an ‘Isa’ relation in the theory of Object-Oriented representation of data. For example, we say a Faculty IsA Employee, a Manager IsA Employee. Here this relation ‘Isa’ implements inheritance.

There are other relations, for example, ‘hasA’, as in a Faculty hasA PersonalInformation, and hasA ProfessionalInformation. The hasA relation is what OOP refers to as an implementation relation and not inheritance.

In an ‘IsA’ relationship, the derived class is clearly of the same kind as that of the base class. For example, a class named PremierCustomer represents an ‘IsA’ relationship with a base class named Customer because a premier customer is a customer. However, a class named CustomerReferral represents a ‘HasA’ relationship with the Customer class because a customer has a customer referral, but a customer referral is not a kind of customer.

Classes in an inheritance hierarchy should have an ‘IsA’ relationship with their base class because they inherit the properties, methods, and events defined in the base class. Classes that represent a ‘HasA’ relationship with other classes are not suited to inheritance hierarchies because they may inherit inappropriate properties and methods. For example, if the CustomerReferral class were derived from the Customer class discussed previously, it might inherit properties that make no sense, such as ShippingPrefs and LastOrderPlaced. ‘HasA’ relationships such as this should be represented using unrelated classes or interfaces.

**Note** that IsA and HasA are not keywords in VB .NET, but merely concepts of OOP.

In this book, we will focus on inheritance and later in this chapter we will cover interfaces for reference. Throughout the book though we will continue to see how inheritance works and is applied in OOP applications. It is crucial to understand the theory of relations and inheritance before you can write code that translates it.

### 5.4 Inheritance Basics

The **Inherits** statement is used to declare a new class, called a **derived** class, based on an existing class, known as a **base** class. Derived classes inherit, and can extend, the properties, methods, events, fields, and constants defined in the base class. The following describes some of the rules of inheritance:
• All classes are inheritable by default. Classes can inherit from other classes in your project or from classes in other assemblies (built-in classes) that your project references. More details later.

• Unlike languages that permit multiple inheritance, Visual Basic .NET permits only single inheritance in classes; that is, derived or child, classes can have only one base or parent, class. A class may be inherited by several other derived classes, for example if class A is the parent of classes B, C and D, then B, C and D cannot inherit any other class, while A is inherited by three classes namely, B, C and D.

• You may prevent a class from being inherited using the **NotInheritable** keyword. We will cover this concept later in this chapter.

5.5 Code Examples

Imagine that you are required to use the Employee class as a base class, and create a Faculty class, and an Administrator (Admin) class as both derived classes from Employee. To do that after you have defined the Employee Class, you start the Faculty class, by using the Inherits statement.

```vbnet
Class Employee
    Dim mFirstName As String
    Dim mSalary As Double
    Dim mDepartment As String
    'other private member variables follow

    Public Event SalaryAboveMillion() As Double

    Public Property FirstName( ) As String
        Get
            Return mFirstName
        End Get
        Set (ByVal Value As String)
        'other code
    End Property

    'and repeat for each property

    Public Sub Promote(ByVal Percent As Single)
        mSalary = Me.Salary * (1 + Percent)
        If MSalary >= 1,000,000 Then
            RaiseEvent SalaryAboveMillion()
        End If
    End Sub

    'other methods definition follows

End Class
```

Now we create the Faculty class, and you realize that a Faculty ‘IsA’ Employee, hence...inheritance!
**Class Faculty**

Inherits Employee 'this statement indicates the Employee is the Parent  

'no code is needed to redefine the properties of Employee here  
'Objects of type Faculty will have the properties, methods and events  
' that Employee owns

End Class

**Note** that for Faculty to inherit Employee, the Employee class must be accessible, in the sense that it must be added to the project (from Project menu, click Add Class) if the source code of the Employee is available. If Employee is only available in compiled version, then you must add a reference to the class (from Project menu, click Add Reference). Note that built-in classes of the .NET framework are all compiled.

**Another Example**

Through inheritance, a child class may use the parent class methods, and change them through a process called overriding; the child class can add new ones if needed. Remember that any member in the parent that is declared **Private** is not accessible in the child or any other class or module code.

The following example defines two classes. The first class is a base class that has two methods. The second class inherits both methods from the base class, overrides the second method, and defines a field named Field.

```vbnet
Class Class1
    Property Color()
        'property code
    End Property
    Sub Method1()
        MessageBox.Show("This is a method in the base class.")
    End Sub
    Sub Method2()
        MessageBox.Show("This is another method in the base class.")
    End Sub
End Class

Class Class2
    Inherits Class1
    Public Field2 as Integer 'new member
    Overrides Sub Method2() 'note same signature as in Class1
        MessageBox.Show("This is a method in a derived class.")
    End Sub
End Class

'Next code is client code:
Sub TestInheritance()
    Dim C1 As New Class1()
    Dim C2 As New Class2()

    C1.Method1() ' Calls Method1 in the base class.
    C1.Method2() ' Calls Method2 from the base class.
End Sub
```
C2.Method1() ' Calls the inherited Method1 from the base class.  
C2.Method2() ' Calls Method2 from the derived class.  
C2.Color = 'some value – Color is defined in the parent  
End Sub

When you run the procedure TestInheritance, you see the following messages:

"This is a method in the base class."
"This is another method in the base class."
"This is a method in the base class."
"This is a method in a derived class."

5.6 The Object Base Class

Almost everything you do with Visual Basic .NET involves objects, including some types not normally thought of as objects, such as strings and integers. All types, including controls in Visual Basic .NET are derived from, and inherit methods from, a base class called Object. The Object class is also the base class, implicitly, of all user-defined classes. This means that anytime you write code to define a new class, then that class implicitly inherits Object, without you having to use the Inherits statement. Do not confuse the class Object with the concept 'object', which is a variable of a type that is a class definition.

This means that any class you define is inheriting all the methods and properties that Object defines. Among these methods is the GetType method, which is used to return the exact type of the current object. This method is useful when grouping objects of different types to determine the actual type of a specific object. For example, if you create several objects based on different types and add them to a combobox control, all these different objects are added as the Object type. Later you can use the GetType method to check the actual type name of each item in the combobox. We will revisit this method once we do Collections in later chapters.

5.7 Inheritance Modifiers

You may see the following keywords used in examples of MSDN or in the IDE designer generated code. Although some of these are topics outside our scope, it is a good idea to review them.

Visual Basic .NET introduces the following class-level statements and modifiers to support inheritance. We will cover some of them later.

- **Inherits** statement — Used in a derived class to indicate that it inherits another class. We have already seen examples of how this is used.

- **NotInheritable** keyword modifier — Prevents programmers from using the current class as a base class. So, if used as in:

  NotInheritable Class President

  Then no other class may inherit President.

- **MustInherit** keyword modifier — Specifies that the current class is intended for use as a base class only. Instances of MustInherit classes cannot be created directly; they can only be created as base class instances of a derived class. Other programming
languages, such as C++ and C#, use the term abstract class to describe such a class. Hence, Student below, may not be a type used to create an object:

MustInherit Class Student

Then this code will cause a syntax error:

Dim O As Student 'no such type name exists

- To prevent exposing restricted items, such as methods or variables, in a base class, the access type of a derived class must be equal to or more restrictive than its base class. Scope Access Modifiers are detailed in the next chapter. Recall that Public is bigger scope than Private. Based on this restriction, a Public class cannot inherit a Private class, and a Friend class – Friend is bigger scope than Private but smaller than Public– cannot inherit a Private class. In short, the derived class scope must be equal or less than the scope of the parent class.

5.8 Me and MyClass or MyBase

You probably have noticed the use of the Me keyword as you were writing code for a form. What is the significance of this keyword, and why is it needed?

When a base class inherits a parent class, and possibly changes the definitions of the some of the parent’s methods, the code may need to refer to the parent’s methods in some cases, while it needs to refer to the child’s methods in others. For example, Promote in Employee may be needed in Faculty. So how can the Faculty class activate Promote from the Employee class in case it is needed? You may use Me or MyClass to refer to methods of the current class, while MyBase to refer to methods in the parent class.

You can use the MyBase keyword to call methods in a parent class when overriding methods in a derived class. For example, suppose you are designing a derived class that includes a method named CalcShipping, that overrides a method inherited from the base class. Overridden methods must have the same name and signature as those they are being overridden by, as we shall examine in the next chapter. The overridden method CalcShipping can call the method in the base class and modify the return value as shown in the following code fragment. Recall that Overriding allows a child class to redefine a method from its original parent class definition. We will detail overriding in the next chapter:

Class ParentClass
  'properties, events and members definition
  Public Function CalcShipping(ByVal Dist As Double, _
                                ByVal Rate As Double) As Double
      Return (Dist * Rate)
  End Function
End Class

Class DerivedClass
  Inherits ParentClass
  Public Overrides Function CalcShipping(ByVal Dist As Double, _
                                          ByVal Rate As Double) As Double
    ' Call the method in the base class and modify the return value.
    Return (MyBase.CalcShipping(Dist, Rate) * 2)
End Class
In the above code, the derived class, DerivedClass is activating the parent class’s CalcShipping by using the MyBase keyword. Hence for a Dist of 4, and a Rate of .2, the value of CalcShipping(4, .2) returned from the child class would be 1.6.

The following list describes restrictions on using **MyBase**:

- **MyBase** refers to the immediate base class and its inherited members. It cannot be used to access **Private** members in the base class. Hence if C is a child of B and B is a child of A, then C’s MyBase refers to members that are not Private accessible from B only. C can not refer to member of A.

- **MyBase** is a keyword, not a real object. **MyBase** cannot be assigned to a variable or passed to procedures.

- **MyBase** cannot be used to qualify itself. Therefore, the following code is illegal:

  ```vbs
  MyBase.MyBase.Method1()  ' Syntax error
  ```

The **MyClass** keyword, on the other hand allows you to call a method implemented in the current class and makes sure that implementation of the method in this class is called rather than an overridden method in a derived class. In other words, **MyBase** means my parent (the base class), and **MyClass** means **Me**. Recall that VB is not case sensitive so **MyClass** is the same keyword as **myclass**. Here are some rules to watch out for:

- **MyClass** is a keyword, not a real object. **MyClass** cannot be assigned to a variable, passed to procedures, or used in an **Is** comparison.

- **MyClass** refers to the current class and its inherited members.

### 5.9 How NEW Works in a Class Hierarchy

Whenever an instance of a class is created, the compiler attempts to execute a procedure named **New**, if it exists in that object class. **New** is a constructor that is used to initialize new objects before any other code in the class executes. We have already covered how constructors work in chapter 3.

When an instance of a derived class is created, the Sub **New** constructor of the base class executes first, followed by constructors in derived classes. For inheritance to work properly, the derived class **must** use a constructor to activate the base class constructor, even if the base class does not have a constructor explicitly written. So make sure the derived class constructor’s the first line of code uses the statement **MyBase.New()** to call the constructor of the class immediately above itself in the class hierarchy.

The **Sub New** constructor is then called for each class in the class hierarchy until the constructor for the base class is reached. At that point, the code in the constructor for the base class executes, followed by the code in each constructor in all derived classes and the code in the last derived class is executed last. For example:
Class Employee
    Sub New( )
    End Sub
    'other code
End Class

Class Faculty
    Inherits Employee
    Sub New( )
        MyBase.New ‘this base is Employee
    End Sub
    'other code
End Class

Class DepartmentChair
    Inherits Faculty
    Sub New( )
        MyBase.New ‘this base is Faculty
    End Sub
    'other code
End Class

If in the client code we have:

    Dim Empx As New DepartmentChair( )

Then New in DepartmentChair is called, which in turn calls Faculty’s New, which in turn calls Employee’s New. You must have this in your code otherwise your class may be ill-designed.

5.10 The Parent Constructor

As you know a child class may access all the Public methods and properties of the parent class. However, this access must be done at the right time. For example, in the following code, the constructor of the child class calls the constructor of the parent. We now know that before that call, none of the parent’s properties are available yet. Hence the line attempting to assign a value to the Color property will cause a syntax error.

Class Class1
    Dim mColor As String
    Public Property Color() As String
        Get
            Return mColor
        End Get
        Set (ByVal X As Value)
            mColor = X
        End Set
    End Property
    'other code
End Class

Class Class2
    Inherits Class1
    Sub New()
MyBase.Color = "Red" 'causes syntax error
   MyBase.New()
End Sub
End Class

The reason is that at the point in the MyBase.Color statement, the parent class is not 'created' yet, as far as the child class is concerned. The code must call MyBase.New first, before attempting to access any of the parent class properties or methods. In fact, the derived class constructor must always call the base class constructor before any other code is added. Hence the following code, even though does not access any of the parent members, still causes a syntax error.

Class Class2
   Inherits Class1
   Sub New()
      Dim a As Integer
      MyBase.New() 'line causes syntax error. VB expects this line first
   End Sub
End Class

Hence the above code should be changed to:

Class Class2
   Inherits Class1
   Sub New()
      MyBase.New()
      Dim a As Integer 'now this is ok
   End Sub
End Class

Note that when calling the parent constructor, the call must match the parent's constructor method. So if the parent Sub New is declared with an integer argument, then the child call must include an integer argument as in the following:

Class Class1
   Sub New(ByVal Y As Integer)
      'some code
   End Sub
   'other code
End Class

Class Class2
   Inherits Class1
   Sub New()
      MyBase.New() 'illegal call since Class1 does not define Sub New()
   End Sub
End Class

Class Class3
   Inherits Class1
   Sub New()
      MyBase.New(25) 'legal since Class1 New expects an Integer passed
   End Sub
End Class
End Sub
End Class

5.11 Parent and Child Type Matching

At this point, it is important to learn how parent and child classes are related through their types. Recall, that a child class, such as Faculty, must call the parent class constructor for proper inheritance. And it is through this inheritance, the child class will contain all the methods and properties of the parent class; in other words, anything the parent class can do, the child class can also do. So in the following code, object O can be used to activate any of the methods and properties of object Q if Faculty was indeed defined to inherit Employee.

Dim O As New Faculty()
Dim Q As New Employee()

In VB, the ability of O to call any of the Employee’s members is translated through conversion using the CType function, and the GetType function.

Recall that the GetType function is inherited from the Object class. Using conversions, an object can be cast into its parent type as the code below illustrates, which uses O and Q objects of the code above:

Messagebox.show(F1(O)) 'this call shows Faculty as the output
Messagebox.show(F1(Q)) 'this call fails
'where F1 is defined below:
Function F1(ByVal x As Faculty) As String
  Return (x.GetType.ToString)
End Function

Note that the function F1 expects an object of type Faculty, hence when O is passed, the code runs as expected. However, when Q is passed, an InvalidCastException occurs due to the fact that Q is of the Employee type. This result is expected. However, would you expect the code to run fine when we change the argument type of the F1 function to Employee?

In fact, the following code succeeds, and shows one message of “Faculty”, then another with “Employee”:

Messagebox.show(F1(O)) 'this call shows Faculty as the output
Messagebox.show(F1(Q)) 'this call shows Employee as the output
'where F1 is defined below:
Function F1(ByVal x As Employee) As String
  Return (x.GetType.ToString)
End Function

The above tells you that an object of a child type, would “match” objects of the parent type, but not vice versa. This is true even if Option Strict is On, which disallows implicit conversions between types, which also brings us to conclude that a conversion from parent to child is a narrowing conversion. This is why Option Strict disallows such conversion because the child class definition may include more members and or modified members that the parent class does not include.
The conclusion is that if you need to write code to manipulate objects, such as the F1 function above, it is better to have the argument’s type as the parent’s as that will allow the calling code to pass objects of the parent type, or any of its child classes.

### 5.12 Handling Parent Events in the Base Class

Derived classes can handle events raised by their base class using the `Handles MyBase` statement. Note that this is different from a base class `object` handling an event defined in the parent class. A derived class object, such as Faculty, can still handle events, such as SalaryAboveMillion that is defined in the parent class. What we are discussing here is the Faculty `class` itself handling an event, so that the event is not raised to the client code. This may be required by the logic, so that possibly a different event is raised, or other statements are executed.

To handle events from a base class, write an event handler in the derived class by adding a `Handles MyBase.EventName` statement to the declaration line of your event handler procedure, where `EventName` is the name of the event in the base class you are handling. For example:

```vbnet
Public Class Employee
    'code to define properties and methods
    Public Event SalaryAboveMillion(ByVal i As Integer)
    'and later ...code to raise SalaryAboveMillion event
End Class

Public Class Faculty
    Inherits Employee
    Sub EventHandler(x As Integer) Handles MyBase.SalaryAboveMillion
        ' Place code to handle this event from Employee here
    End Sub
End Class
```

With the code above, an object of type Faculty will NOT be able to ‘see’ nor handle the SalaryAboveMillion event because the class itself is handling it. This way, Faculty controls how to handle the event, and may raise a different event, or execute other statements instead. Hence if the intention is to allow an object of type Faculty to handle event SalaryAboveMillion, then the EventHandler procedure above should **not** be written.

### 5.13 Button Control Click Handler

The button control, click event handler, has two arguments; the first name is named sender and is of type object. This argument will represent the object (the button) that caused the event to fire. As we shall see in this section, you can use this argument to write one click event handler for a number of buttons. This way, if the user clicks one of the buttons, you can check the `s` argument to figure out, which button on the form caused the event Click to fire, hence the handling code is run accordingly.

In the handling code, you can check the `Sender` object for its name, hence you can write different handling code for each button based on the name:

```vbnet
If Sender.Name.ToString = “GiveRaise” Then
```
Using the same argument, you can write one handler for many controls, and using the Sender.GetType you can write handling code for buttons vs. images, for example.

In handling multiple events in the same handler, don’t forget to list each objectName.EventName after the Handles clause separating each Name.EventName pair with a comma as in:

Handles Employee.GiveRaise, Faculty.GiveRaise, AnyObject.EventName

In the next chapter, we will cover topics that may arise when working with base classes, such as overriding, shadowing and overloading. This will help us develop more robust child classes.

**Exercises**

5.1 Based on the classes developed in the exercises of chapter 4, develop a GreenCar that uses Car as a parent.
   
   a. GreenCar will suppress the parent’s GasLow event so that objects based on GreenCar will not be able to respond to the GasLow as is. Instead, when GasLow fires, GreenCar will raise its own GasLow – you may use a different name- but only if the gas is at 2 or less gallons. We will do more with GreenCar once we cover the next chapter.

5.2 On the client side, divide the form into two areas, one for Car and another for GreenCar. Make sure each area has the needed controls to enter each car’s data, with two sets of buttons to Drive, and Pump Gas. However, since both sets of Drive and Pump Gas buttons will do the same action; handle these events in one handler per button. For example handle Car and GreenCar’s Drive click events in one handler, and do the same for the Pump Gas buttons. This will require you check which button was clicked so that you know which object to work on.
Chapter 6 Overloading, Overriding, Shadowing and Access Modifiers

In this chapter we will add finishing touches to Inheritance that give it its power. We specifically cover method overriding – which changes how a method works from parent to child, shadowing, which is similar to overriding but applies to more member types, overloading – which allows callers to call the same method name but with different argument list and types. We will also cover the different Access or Scope modifiers to hide or enable access to elements from parent to child and other classes in an application.

6.1 Inheritance-Based Polymorphism

The word Poly -Greek origin- means the many faces of. Polymorphism is a powerful aspect of OOP. It means that a method may have different faces. Huh? Yes, for example, you may Print to the printer, or Print to the screen, or even Print to a file. Here we are using the word Print in many contexts; we are using the same word but the actual task is different. This makes it easy for others who are re-using our code to understand and use the methods in our classes much easier. For example, imagine if you need to print to the printer, then you must say PrinterPrint, where if you needed to print to the screen, then you say ShowOnScreen, and to print to a file you had to say CopyFile. With polymorphism, you may say Print(Printer), Print(Screen), and Print(File), which is much easier to remember, where Printer, Screen and File are different types that may not be related.

Note: Although Polymorphism is not restricted to Inheritance, it is greatly appreciated there.

Suppose that a Faculty gets raises at a different rate than regular employees or Admins. In that case, the Promote method defined in the Employee class, must be 'redefined' in the Faculty class, and again in the Admin class. You can surely use a different method name in child classes, but that makes it harder on the client programmer to remember what the method name is. So using Promote as the method name so that Promote in Employee works one way and Promote in Faculty works differently is one aspect of polymorphism- using the same name but it does different things in different classes. In VB terms, you need to use overriding to implement such a method- same method name but different code to execute across parent-child classes.

On a different level of Polymorphism, you may want the client programmer to pass the percent raise as a Single type, Double or Decimal. If you write the Promote method to receive a Single type argument, you're not giving the programmer the choice. If you like, you can use 'many versions' of the Promote method so that it may be called on a Single, Double or Decimal. The same method works on different data types is another face of polymorphism that is implemented in VB using overloading. Overloading may be implemented in the same class and or across base and derived classes. The following is an example:

```vbnet
Overloads Public Sub Promote(ByVal Percent As Single)
    'note the Single Type.
    ' more on the Overloads keyword later
    Me.Salary = Me.Salary * (1 + Percent)
End Sub
```
Overloads Public Sub Promote(ByVal Percent As Double)  
'note the Double data Type  
'the code below may be different from the above version  
    Me.Salary = Me.Salary * (1 + CSng(Percent))  
End Sub

The next code is not allowed because the argument list is the same as the one provided above.

Overloads Public Sub Promote(ByVal Rate As Double)  
'note the signature is a copy of the above Promote  
'some other statement

End Sub

When you overload a Sub or Function procedure, you must either use a different number of arguments and or different data types. Changing the variable name in an argument does not constitute a difference in the argument list.

In the above, we are overloading Promote in the first and second versions so that the client code could pass a percentage as a Single or Double declared variable, allowing for greater flexibility in the client code. This is used heavily in Forms and Controls classes. The third version of the Promote method uses the same signature of the second version which causes a syntax error. More details on rules of overloading later in this chapter.

Note how in the class code Salary stores the actual Salary value as a Single variable, then to allow for later-on changes in the class to be invisible to the client code, the overloaded methods take different data type arguments representing the Percentage, and convert that to whatever the class uses. Say after a year, the Employee class developer realizes that the Salary should be represented by a different data type then all the clients using this class do not have to change their code accordingly.

The class makes this change invisible, which is an extremely desirable feature - Imagine Microsoft changing how forms work, hence changes the Class Form, and then you have to change all your applications that were based on the 'old' class Form; that would be very costly.

In Summary, Polymorphism allows for greater flexibility in client code. It does mean the class definition would be bigger than before, but that's a very small expense compared to what the client code can do. Now if overloading allows for flexibility for the client, what if the child class needs to overload Promote? Can it do that? No. In the same class, you can overload, but from parent to child, the child class has to OVERRIDE.

Overriding is Polymorphism in inheritance. Here when a derived class wants to redefine a Sub or Function that the child inherited from the Base class, the child needs to use the Overrides keyword in the child sub or function definition. In addition, the base class must indicate the sub or function as Overridable, meaning that a derived class may override it. The code below illustrates.

For example, you could define a class, BaseTax, that provides a baseline functionality for computing sales tax in a state. Classes derived from BaseTax, such as CountyTax or CityTax, could implement methods such as CalculateTax as appropriate.
Polymorphism comes from the fact that you could call the CalculateTax method of an object belonging to any class derived from BaseTax, without knowing which class the object belonged to. The Test procedure in the following example demonstrates inheritance-based polymorphism:

```
Const StateRate As Double = 0.053 ' 5.3% State tax
Const CityRate As Double = 0.028 ' 2.8% City tax

Public Class BaseTax 'this is the base class
    Overridable Function CalculateTax(ByVal Amount As Double) As Double
        Return Amount * StateRate ' Calculate state tax.
    End Function
End Class

Public Class CityTax 'this is a derived class
    Inherits BaseTax
    Private BaseAmount As Double

    ' This method below calls a method in the base class
    ' and modifies the returned value
    Overrides Function CalculateTax(ByVal Amount As Double) As Double
        Base = MyBase.CalculateTax(Amount)
        Return ((CityRate * (Base + Amount)) + Base)
    End Function
End Class

'Next is the client code
Sub Test()
    Dim Item1 As New BaseTax()
    Dim Item2 As New CityTax()
    MessageBox.Show(Item1.CalculateTax(12.4).ToString) ' base purchase
    MessageBox.Show(Item2.CalculateTax(12.4).ToString) ' city purchase
End Sub
```

In this example, Item1 and Item2 both use the same method name, CalculateTax, but each item returns a different result. The programmer of Test need not worry about different names of methods to call; only one name to remember, namely, CalculateTax.

### 6.2 Overloading Properties and Methods

Overloading is the creation of more than one procedure, constructor, or property in a class with the same name but with different argument types. Overloading is especially useful when your application logic dictates that you use identical names for procedures that operate on different data types. For example, a class that can display several different data types could have a `Display` procedure that looks like the following:

```
Overloads Sub Display(ByVal theChar As Char)
    ' Add code that displays Char data.
End Sub
```
Overloads Sub Display(ByVal theInteger As Integer)
    ' Add code that displays Integer data.
End Sub
Overloads Sub Display(ByVal theDouble As Double)
    ' Add code that displays Double data.
End Sub

Without overloading, you would need to create distinct names for each procedure, even though they do the same thing, as shown next:

Sub DisplayChar(ByVal theChar As Char)
    ' Add code that displays Char data.
End Sub
Sub DisplayInt(ByVal theInteger As Integer)
    ' Add code that displays Integer data.
End Sub
Sub DisplayDouble(ByVal theDouble As Double)
    ' Add code that displays Double data.
End Sub

Overloading makes it easier on the client code because the client code programmer is given a choice of data types that can be used. For example, the overloaded Display method discussed previously can be called with any of the following lines of code:

    Display("7"C)  ' Call Display with type Char
    Display(7)    ' Call Display with type Integer
    Display(7.4R) ' Call Display with type Double

At the run time, the compiler calls the correct procedure based on the data type of the parameters specified in the calling statement. If you do not appreciate overloading, imagine if you have a String and a Double and you need to convert them to Integer. The CInt function would be CIntSTR and CIntDBL, and not one function CInt.

Even though overloading is not directly related to inheritance as a topic, however it is heavily used in child classes.

6.2.1 Overloading Rules

The general rule to follow when overloading, is that the method or property name must be the same, but the argument list must be different. You can create an overloaded member for a class by adding two or more properties or methods with the same name. Except for overloaded derived members, each overloaded member in the same class must have different argument lists, and the following items cannot be used as a differentiating feature when overloading a property or procedure:

- Modifiers, such as ByVal or ByRef, that apply to a member or parameters of the member.
- Names of parameters or arguments.
Note that the **Overloads** keyword is optional when overloading, but if any overloaded member uses the **Overloads** keyword, then all other overloaded members with the same name must also specify this keyword.

The following example creates overloaded methods that accept either a String or a Decimal dollar amount and return a Decimal containing the total tax. To use this example add the code below to the Employee application.

```vbnet
Public Class IncomeTax
    Overloads Function TaxAmount(ByVal decInc As Decimal, ByVal TaxRate As Single) As Decimal
        Return (decInc * TaxRate)
    End Function
    Overloads Function TaxAmount(ByVal strInc As String, ByVal TaxRate As Single) As Decimal
        Return (CDec(strInc) * TaxRate)
    End Function
End Class
```

Add the following statements to a Click handler of a Button on the form:

```vbnet
Const TaxRate As Single = 0.125 '12.5% tax rate
Dim strInc As String = "64000.00" 'income amount as a String
Dim decInc As Decimal = 64000.00 'income as a Decimal
Dim test1 As New IncomeTax()
'Call the same method with two different kinds of data.
'You may want to add ToString conversion
MessageBox.Show(test1.TaxAmount(strInc, TaxRate))
MessageBox.Show(test1.TaxAmount(decInc, TaxRate))
```

### 6.3 Overriding Properties and Methods in Derived Classes

By default, a derived class inherits methods from its base class. If an inherited property or method needs to behave differently in the derived class it can be overridden; that is, you can define a new implementation of the method in the derived class. The following modifiers are used to control how properties and methods are overridden:

- **Overridable** — Allows a property or method in a base class to be overridden in a derived class.

- **Overrides** — Overides an **Overridable** property or method defined in the base class. This keyword is added in the derived class.

- **NotOverridable** — Prevents a property or method from being overridden in a derived class. Public methods and properties are **NotOverridable** by default.

Overriding is needed in a derived class if an inherited member or more cannot be used "as is". For a derived class to override members, the base class must specify those members as **Overridable**, since by default Public methods and properties in the base class are **NotOverridable**. In that case —overriding is not possible— Shadowing is an alternative.
In practice, overridden members are often used to implement polymorphism. The following rules apply when overriding methods:

- You can only override members that are marked with the `Overridable` keyword in their base class.
- Properties and methods are `NotOverridable` by default.
- Overridden members must have the same arguments as the inherited members from the base class.
- The new implementation of a member can call the original implementation in the parent class using the `MyBase` keyword before the method name.

The following example illustrates the points above. Suppose you want to define classes to handle payroll. You could define a generic Payroll class that contains a `RunPayroll` method that calculates payroll for a normal week. You could then use Payroll as a base class for a more specialized BonusPayroll class, which could be used when distributing employee bonuses. The BonusPayroll class can inherit, and override, the `PayEmployee` method defined in the base Payroll class.

In the code below we define a base class, Payroll, and a derived class, BonusPayroll, which overrides an inherited method, PayEmployee. A procedure, `RunPayroll`, creates and then passes a Payroll object and a BonusPayroll object to a function, `Pay`, that executes the `PayEmployee` method of both objects.

```vbnet
Class Payroll
    Overridable Function PayEmployee(ByVal HoursWorked As Decimal, ByVal PayRate As Decimal) As Decimal
        Return (HoursWorked * PayRate)
    End Function
End Class

Class BonusPayroll
    Inherits Payroll
    Dim BonusRate As Decimal = 1.09
    Overrides Function PayEmployee(ByVal HoursWorked As Decimal, ByVal PayRate As Decimal) As Decimal
        ' The following code calls the original method in the base class, and then modifies the returned value.
        ' note that both functions have the same signature
        Return (MyBase.PayEmployee(HoursWorked, PayRate) * BonusRate)
    End Function
End Class
```

Then in the client code add the following statements under a Click event handler of a Button:

```vbnet
Dim PayDay As New Payroll()
Dim BonusPay As New BonusPayroll()
Dim HoursWorked As Decimal = 32.0

MessageBox.Show("Normal pay is: " & _
    PayDay.PayEmployee(HoursWorked, PayRate))
MessageBox.Show("Pay with bonus is: " & _
    BonusPay.PayEmployee(HoursWorked, PayRate))
```
A final note on overriding, is that if the parent class has several overloaded methods, the the child class may override and overload those methods. Hence in the child class you will notice the Overrides and Overloads keywords used in a method definition. In addition, a child class may override one or all versions of the method in the parent class as shown below.

```vbnet
Public Class IncomeTax
    Overridable Function TaxIs(ByVal decInc As Decimal) As Decimal
        Return (decInc * 1.09))
    End Function

    Overridable Function TaxIs(ByVal strInc As String) As Decimal
        Return (CDec(strInc) * 1.09)
    End Function
End Class

Public Class MoreTax
    Inherits IncomeTax
    Overrides Overloads Function TaxIs(ByVal decInc As Decimal) As Decimal
        Return (decInc * 1.15))
    End Function

    Overrides Overloads Function TaxIs(ByVal strInc As String) As Decimal
        Return (CDec(strInc) * 1.15)
    End Function
End Class
```

Now that we covered overloading and overriding, it is important to cover in detail, scope or access modifiers. Access modifiers affect what parts of an application have access to a declared variable, method or property.

### 6.4 Access Modifiers

When declaring variables, where you put the declaration statement decides on the scope of that variable. However, you may change (increase or restrict) that scope by using a set of keywords known as **Access Modifiers**. It is important here to note that access modifiers are also used on Sub and Function definitions to make them accessible from one module to another, as an example.

The following list shows the different access modifiers available in VB:

- **Private**
- **Public**
- **Friend**
- **Protected**
- **Shared**

**Private**
To declare a Private variable, you may omit the Dim keyword as use:
Private x As String

Variables declared inside a Sub or Function procedures are private by default. These variables are not available to any other coding element outside the Sub or Function procedure they are declared in. On the global (form, module or class) level, elements for which you declare `Private` are available for reference to every procedure in that form, class or module, but not to any code in a different form or class or module. A module is a file that you can use to add code (Variables, Subs and Functions only) that you may want to be available to other forms and or classes in your project.

The `Dim` statement at the module level defaults to `Private` if you do not use any accessibility keywords. Hence:

Dim x As String

Is the same as:

Private x As String

However, you can make the scope and accessibility more obvious by using the `Private` keyword in the declaration statement. It is good programming practice to explicitly write in code what otherwise is the default.

**Note** that when you declare a variable inside a Sub or Function, then that variable has local (to that Sub or Function) access only, and may not be modified by `Private` or `Public` keywords. Access modifiers are normally used on the module level.

**Public**
The `Public` keyword is easily viewed as the opposite of `Private`. It makes a variable or a method (Sub or Function) accessible to all components (all files) in the Project. This is used when you have a constant value that you want to use throughout a project, or other projects in the current solution, for example:

Public Const Pi As Single = 3.14

Another example where `Public` is used, is in cases where you have a Function or a Sub procedure that other modules in the current project or other projects in the current solution need access to.

**Note** that when you declare a Sub or Function with the `Private` keyword, then that method is accessible only within the module it is written in.

**Note** also that when you use the `Public` keyword on a member that is declared inside a class (remember a form is a class), then that member (whether Function, Sub or variable) is only available after an object based on that class is instantiated. Hence to make a member public to other files in a project or solution, you need to declare them `Public` in a code module, not a class. For example, P in the following code is not available to other files in the current project:

```vbnet
Class C
    Public P As Integer
End Class
```
Any form or module in the current project cannot use P directly, unless an object based on C is created:

```vba
Dim O As New C
O.P = 25
```

The following code on the other hand is not allowed outside class C:

```vba
P = 25 'since P does not exist on its own
```

**Friend**

This keyword is used to provide access to the item it applies to (variable, object, or methods) to all the components in the current project, but not other projects. **Friend** is looked at, as friends are in real life. You discuss the matters of your family (current project) with friends, but not outsiders (other projects).

**Protected**

Protected members are available from parent to child classes only. This is a very desirable feature since sometimes you have variables, and or methods that you do not want to expose to the outside world, but need your children to be able to use. Hence a class that declares members as **Protected** is allowing any derived class to use those members. Other classes in the current project are not allowed to use those **Protected** members.

**Shared**

By default, class data is specific to the *instance of the class only*, but there may be occasions when you want a data item to be available and shared by all objects created from that class. The shared value, if set once, is available to all instances of the class. In such cases, you can use the **Shared** modifier to cause a variable to share the same value in all instances of a class. Shared members are sometimes referred to as "static members" in other programming languages.

You can call shared methods directly using a class name without first creating an instance of the class. For example, in the math library, you have been using the Format function, without having to create an instance of the math class! The same goes for the MessageBox class, and its Show method. This method was defined using the **Shared** keyword, meaning we can call Show without having to create an instance of its corresponding class, MessageBox, as opposed to the Promote method for example, which is unavailable to client code before we create an object based on the Employee class.

Shared members are used in two ways:

1. Shared variables can be set so their value is applied to all objects based on the class they are defined in.

2. Shared methods can be activated without needing to create objects of the class the shared method is declared in. This is mostly common in libraries such as Math and Graphics.

The following is an example of a shared variable:

```vba
Class Employee
    Shared Limit As Single
    ...
```
Property Salary( ) As Single
    Set(ByVal Value As Single)
    If Value > Limit Then
        RaiseEvent SomeEvent
    ....
    End Property
End Class

Then in client code we can have:

Sub Button1_Click(...) Handles Button1.Click
    Employee.Limit = 250000
    'this way all object based on Employee we create thereafter
    'will have the limit set to 250000
End Sub

Dim WithEvents x As New Employee
Dim WithEvents y As New Employee

'Later:
x.Salary = 150000  'event will not be triggered
y.Salary = 300000  'event will be triggered

In the above code, Limit is applied to both x and y objects, thus allowing the client code to set the limit once and have it applied to all objects based on Employee.

6.5 Shadowing Through Scope

When two programming elements share the same name, one of them can hide or shadow the other one. In such a situation, the shadowed element is not available for reference; instead, when your code uses the name, the compiler resolves it to the shadowing element.

For example, when you declare a variable A at the module level, and then later declare another variable A at a local level, then while the local A is visible, the module level A may not be referenced. In this case, the local A –and its content- shadows the module A and its content. So the rule is the element with the narrower scope shadows the other element (block scope is the narrowest).

In the following example, a module defines a Public variable named Temp, and a procedure within the module declares a local variable also named Temp. References to Temp within the procedure access the local variable, while references to Temp outside the procedure access the Public variable. In this case, the local variable Temp shadows the module variable Temp. When it is shadowed by a local variable, a module variable is still accessible from within a procedure by qualifying the variable with the name of the module, class, or structure it is declared in, as illustrated in the following example:

Module Mod4
    Public Temp As Integer ' This Temp is a module-level variable.
Sub UseModuleLevelTemp( )
  Temp = 1 ' Set module-level Mod4.Temp to 1.
  ' ...
  Show( ) ' Call Show, which displays module-level Temp.
End Sub

Sub Show( )
  Dim Temp As Integer ' This Temp is a local variable.
  Temp = 2 ' Local Temp has a value of 2.
  MessageBox.Show(CStr(Temp)) ' this shows 2
  MessageBox.Show(CStr(Mod4.Temp)) ' Mod4.Temp has a value of 1.
End Sub

End Module

An element can shadow another element in another way. Rather than shadowing through scope, an element can be shadowed through inheritance. This happens when a deriving class redefines a member of a base class, in which case the shadowing is done through inheritance.

If a derived class redefines a programming element inherited from a base class, the redefining element shadows the original element. What's more interesting is that you can shadow any type of declared element, or set of overloaded elements, with any other type. For example, an Integer variable can shadow a Function procedure. If you shadow a procedure with another procedure, you can use a different argument list and a different return type. This is useful if elements in the parent class are not declared as **Overridable**.

Note that if the shadowing element is not accessible from the consuming code, for example if it is declared Private, shadowing is defeated and the compiler resolves any reference to the same element it would have if there had been no shadowing. If the shadowed element is a procedure, the resolution is to the proximate accessible version with the same name, argument list, and return type.

Do not confuse shadowing with overriding. Both are used when a derived class inherits from a base class, and both redefine one declared element with another. But there are differences between the two. The following table compares shadowing with overriding:

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Shadowing</th>
<th>Overriding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Level</td>
<td>Any Access Level allowed</td>
<td>Cannot expand the access level of the base element</td>
</tr>
<tr>
<td>Keyword used-</td>
<td>Base element does not require any keyword; derived element may use <strong>Shadows</strong> but it is optional</td>
<td><strong>Overridable</strong> keyword required in base element; then derived element must use <strong>Overrides</strong></td>
</tr>
<tr>
<td>required/optional</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When you access an element from a derived class, you normally do so through the current instance of that derived class, by qualifying the element name with the **Me** or **MyClass** keyword. If your derived class shadows the element in the base class, you can access the base class element by qualifying it with the **MyBase** keyword.
6.6 ByRef VS ByVal

Now that we know more about objects, we need to cover one more related topic. You may be familiar with the keywords **ByVal** and **ByRef**, which are used in the signature of a Sub procedure or a Function. You may not know that their use (or lack of) may be affected by the data type of the argument in question. How an argument is passed is related to the category of the data type of the variable.

Variables can be of the **Reference** type or of the **Value** type. A variable is said to be a reference type if the variable is an object type such as String, Employee, Form and all controls as opposed to a fundamental or value type such as Integer, Single, Double, or Boolean among others. In short when an argument is a reference type, passing it will always be done by reference even if the **ByVal** keyword is used. Only value-type arguments can be passed by value or reference.

In this section we explain how passing arguments work, and why their type affects whether they are passed by value or reference.

When you write a user-defined procedure, your goal is to call that procedure a number of times to make it run the code it contains. For example:

```vbnet
Sub DisplayInfo(X As Integer)
    MessageBox.Show(X.ToString, "My Application")
End Sub
```

Where X is a variable that is being passed to the procedure. Passed from where? From the calling code, for example:

```vbnet
Sub zz(a,b) Handles Button1.Click
    Dim Y As Integer
    'some code to process data into Y
    Call DisplayInfo(Y)
End Sub
```

Here Y contains the data we want to pass to DisplayInfo into X. View it this way, the Call statement above represents a trip to Sub DisplayInfo. Variable Y in the Call statement is a bag that will be sent along the trip, so we place items in that bag, and hand the bag over to Sub DisplayInfo.

Now, how this bag is being exchanged is called Variable Passing. And the idea is, do we copy what's in the bag before we leave zz, so the original bag stays intact, or do we hand the bag to DisplayInfo, where DisplayInfo may actually change the content of that bag. The first method (making the copy) is called passing by value using the keyword **ByVal** in VB, and the second is called passing by reference using the keyword **ByRef**.

When passing by value, the content of the variable in the calling procedure (variable Y in procedure zz above) will be copied into RAM and the copy of the content is passed to DisplayInfo, therefore the value (content) of Y is passed to DisplayInfo into X. On the other hand, when you pass by reference, the location of the passed variable is passed to the called procedure; that is the location of Y in RAM is passed to DisplayInfo rather than its content, where the location in RAM is referred to in programming as the Reference, hence passing by reference. So when passing by value, X will contain a copy of Y's content, while when passing by reference, X will contain the address of Y in RAM.
Note that zz is the code that passes the variables, but DisplayInfo is the one that decides on the passing mechanism. Sub procedure zz, which is called the "calling" procedure, passes the content or the reference to the variable based on DisplayInfo, which is called the "called" procedure. This is the basic idea. As you see, passing by value uses more RAM, by duplicating the variable Y, while passing by reference uses less RAM but may be confusing to the coder of zz, since Y before the call "may be" difference after the call. So which method should you use, and do we have a choice?

Let’s examine how each method works through the example below. To pass a variable by value, the signature line of the procedure must use the **ByVal** keyword as in:

```vba
Function ValidateData(ByVal Data As Integer) As Boolean
  If Data <= 0 Then
    Return False
  Else
    Return True
  EndIf
End Function
```

In the above function, Data is an Integer that will receive a copy of the data passed to the function ValidateData. In this case, the logic of ValidateData does not change Data (the variable), hence whether Data was passed by value or reference does not make any difference. However, if we change the logic of ValidateData, then the picture may look a bit different, as in:

```vba
Function ValidateData(ByVal Data As Integer) As Boolean
  If Data <= 0 Then
    Data = 0
    Return False
  Else
    Return True
  EndIf
End Function
```

In the above updated version of ValidateData, the function changes the content of Data if it contains a value less or equal to zero by assigning Data to 0. So if the calling code passed -5 into Data, then during the function call, Data will change its content to 0. In the case above, Data is passed by value, so this will have no effect on the original passed variable, but if we change to the following code, the outcome will be different:

```vba
Function ValidateData(ByRef Data As Integer) As Boolean
  If Data <= 0 Then
    Data = 0 'line causing change
    Return False
  Else
    Return True
  EndIf
End Function
```

And if we call the above as in:

```vba
Dim x As Integer = -5
Messagebox.show(ValidateData(x).ToString) 'this will show False
Messagebox.show(x.ToString) 'this will show 0 – unexpected!
```
The above code demonstrates that ValidateData has changed the content of variable x due to line marked “line causing change”. In this case variable x's location is passed into Data, so Data in the function is “pointing” to variable x. It’s as if the line Data = 0, is the same as saying x = 0.

But what about reference types? In the above example we learned how a variable like x may be affected by a Function or a Sub procedure call based on whether the signature uses **ByVal** or **ByRef** keywords. Variables of data types such as Short, Integer, Long, Single, Double, Decimal, Boolean will follow the same pattern. These types are called **Value Types**. Types such as String, arrays of any type, and any user defined type –which inherits from the Object class implicitly, are called **Reference Types**. These types will be passed by reference, even if the **ByVal** keyword is used. The example that follows explains why.

When a Value Type variable is declared in RAM, a number of bytes is reserved for that variable, and that block of RAM is referenced using the variable name. However, when a Reference Type variable – usually referred to as object - such as Array of Integer, is defined, then 4 bytes for each location in the array is reserved in RAM. If the array contains say 20 Integer locations and is called NUMS, then 20 4-byte blocks of RAM are reserved for NUMS. You know from your previous programming experience that NUMS(0) refers to the first location in the array, and NUMS(1) refers to the second location as so on. But what does NUMS refer to? In other words, if we look at RAM directly, what does NUMS contain? It contains the starting address of the first Byte of NUMS. Hence NUMS itself contains an address or reference to a RAM location, hence the reason why arrays are called Reference Type, regardless of the type of the array.

The same applies to String, and all user-defined types or any class that inherits from **Object**. But how does this affect the way we pass variables of a Reference Type? To understand how passing Reference Types works, consider the following example:

```vbnet
Dim NUMS() As Integer = {4,7,3,9,-10}
Messagebox.show(ValidateArray(NUMS).ToString) 'this will show False
'Just checking whether the original array NUMS was changed during the call:
Messagebox.show(NUMS(0).ToString) 'this will show 0 instead of 4

Function ValidateData(ByVal Data() As Integer) As Boolean
    Dim I As Integer
    For I = 0 to Data.GetUpperBound(0)
        If Data(I) <= 0 Then
            Data.Clear() 'this line causes change to the original array passed
            Return False
        EndIf
        Next I
    Return True 'means that all elements of Data are >0
End Function
```

If you are thinking, but the array was passed By Value, hence no change should apply to NUMS, then just think about what does NUMS contain, and what was copied into Data as a result of the call.

In the above code, NUMS was declared as an array, so NUMS itself contains a RAM address. The call to the function uses **ByVal** so the content of NUMS (a RAM location) is copied into Data. Now, line 4 in the function resets the array via the Clear array built-in function. But which array? The one whose address is copied into Data, and that address is the address of NUMS. Hence,
even though you passed NUMS by value, its content has changed because the placeholder in the called function contains the address (not the value) of the original argument. This may sound frustrating, but you may use it to your advantage. Below are more examples, and some design ideas.

In the following code:

```
Dim x As New ObjectType 'where ObjectType inherits from Object
```

Let's say the compiler reserved 24B in RAM for x. The first byte is the location at say ABD9823. That location is copied into a variable named x. So now the actual content of x is ABD9823. So, if you pass x `ByVal` the compiler will copy the content of x into another variable, say y. Now the content of y is ABD9823. If you on the other hand pass x by reference, the compiler will pass whatever x points to, into y and that content is still ABD9823; hence `ByVal` or `ByRef` results in the same outcome.

How then do you ensure that if you pass an object to another method, then its content is not going to change? You have two solutions. If you are on the caller's side, you start your code by creating an object of the same type as the passed variable then you copy each property value of the original object then pass the copy. Here is an example:

```
Dim myobject As New ObjectType(.....)
'more code to assign myobject properties to values
```

Later to pass myobject to another method use the following code to create a copy of myobject into another object x of the same type then pass x.

```
Dim x As New ObjectType()
x.property1 = myobject.property1
x.property2 = myobject.property2
'and so on, then you pass x
```

If you are at the receiving end, start the first line of code by creating the new object, and repeat the above steps:

```
Sub Method1(ByRef SomeObject As ObjectType)
  Dim y As New ObjectType
  'same as above, copy each property value
  'other code
End Sub
```

Note that in the following code, both x and y point to the same RAM location, hence y is not a 'different' copy of x. Both y and x point to the same object:

```
Dim x As New ObjectType()
Dim y As New ObjectType()
'code to assign x's properties
y = x 'y points to whatever x points to; both point to the same RAM location
```
6.6.1 Choice of the Passing Mechanism

In choosing between the two passing mechanisms, the most important criterion is the need of the passed variable for change. The advantage of passing an argument *ByRef* is that the procedure can return a value to the calling code through that argument. This is useful in cases where a function needs to return a number of values back to the caller, and each value is a different type. You cannot return an array since the array elements have to be of the same type, hence use the argument passed and assign it the return value.

The advantage of passing an argument *ByVal* is that it protects a variable from being changed by the procedure.

Although the passing mechanism can also affect the performance of your code, the difference is usually insignificant. One exception to this is a Value Type passed *ByVal*. In this case, Visual Basic copies the entire data contents of the argument, and therefore, for a large type such as a structure (not covered in this book), it is more efficient to pass it *ByRef*. A structure is like a class definition but you cannot inherit a structure.

In summary we have the following facts:

1. The data types that the compiler defines are two types. The *Fundamental* data types, such as Integer, Single, Date, Boolean, etc. The other type is called *Reference* data types such as String, any array of any data type, and the Object type.

2. All user-defined classes inherit from the Object Class.

3. All types that are derived from (inherit) the Object type are passed by reference. This is why they are referred to as Reference types.

6.7 Interfaces and Namespaces

Before we leave this chapter, there are two more topics requiring coverage that is needed in the chapters to come. An *Interface* is code that declares (note declares but not fully codes) properties, methods, and events that classes can implement. Interfaces allow you to state what features a class which ‘implements’ the interface should have. Hence, an interface by itself is not useable as a type, rather it provide a listing of the members of what a future class based on the interface should have. A *Namespace* on the other hand, is a way for developers to group related classes and or interfaces together, much like a file cabinet.

6.7.1 Interfaces

Interfaces are like empty classes in that they only declare properties, methods, and events. Unlike classes though, interfaces do not provide the implementation of these members. A class that uses an interface through the *Implements* statement may then implement the members that were declared in the interface used in the Implements statement. The example below will clarify.

To create an interface, add the code sample shown below. It must begin with the *Interface* keyword and the name of the interface, and must end with the *End Interface* statement. It is common to precede the name of interfaces with an uppercase i.
Interface IExample

End Interface

Next you need to add statements that declare the properties, methods, and events the interface supports. For example, the following code defines one function, one property, and one event.

Interface IExample
    Event Warning(ByVal Complete As Boolean)
    Property Words() As String
    Function GetIndex() As Integer
End Interface

The next step is to implement an interface. If the interface that you are implementing is not part of your project, add a reference to the application containing the interface by right clicking the project icon and choosing the Add Reference option, then point to the project name.

To implement an interface, create a new class that uses the Implements statement as shown below. The Implements statement is made of the Implements keyword followed by the interface name.

Class Code
    Implements IExample
End Class

Next add code that declares and defines every member that the interface declares. Hence in the example above, the class Code must add statements to define the Words property, the GetIndex function, and somewhere raises the Warning event. Each of these members must use the Implements statement that lists which member of the IExample is being implemented.

Class Code
    Implements IExample
    Public Event Warning(ByVal C As Boolean) Implements IAsset.Warning
    Private divisionValue As String
    Public Property Words() As String Implements IExample.Words
        Get
            Return divisionValue
        End Get
        Set(ByVal value As String)
            divisionValue = value
            RaiseEvent Warning(True)
        End Set
    End Property
    Private IndexValue As Integer
    Public Function GetIndex() As Integer Implements IExample.GetIndex
        Return IndexValue
    End Function
    Public Sub New(ByVal Division As String, ByVal IND As Integer)
        Me.divisionValue = Division
        Me.IDValue = IND
    End Sub
End Class
In addition to implementing the elements of the Interface, the implementing class may also add new features at any time by developing additional interfaces and implementations. Hence unlike multiple inheritance which is not supported in VB, where a class may not use the Inherits statement more than once, a class may implement multiple interfaces, using the Implements statement a number of times.

Some programmers may argue that interfaces are useless, while others view them as conceptual designs. The focus of this book is not on how valuable interfaces are nor on strategies on creating interfaces, but rather on understanding the concept, as we will cover some built-in classes that use interfaces in later chapters.

6.7.2 Namespaces

Namespaces are not related to interfaces conceptually, but it would be useful to cover this topic at this point now that we have covered classes and interfaces. A namespaces is used as an organizational system as it provides a way of classifying programming components such as classes and interfaces. Note that a namespace does not define a type like a class does, hence you cannot declare an element to have the data type of a namespace.

The **Namespace** statement declares the name of a namespace and causes all the code the follows it to be compiled within that named-space. Examples below provide details. The following code defines a new class in the NS namespace:

```vbscript
Namespace NS
    Class MyClass
        'code to declare and define members of MyClass
    End Class
End Namespace
```

The example below declares two namespaces, one nested inside the other. Typically Class A and Class B are logically related.

```vbscript
Namespace NS1
    Namespace NS2
        Class A
            ' class code
        End Class
        Class B
            ' class code
        End Class
    End Namespace
End Namespace
```

The following example declares multiple nested namespaces on a single line, and is equivalent to the previous example.

```vbscript
Namespace NS1.NS2
    Class A
        ' class code
    End Class
    Class B
        ' class code
```
The following example accesses the class defined in the previous examples.

```vbnet
Dim NewObject As New NS1.NS2.A
```

Later in the code, any call to a method or property of A, must be preceded by NS1.NS2. In order to make the reference to class A and its members shorter, the **Imports** statement may be used, so that the declaration above becomes:

```vbnet
Imports NS1.NS2
Dim NewObject As New A
```

You may also define several namespaces inside one, as the following example shows.

```vbnet
Namespace MainNS
    Namespace NS1
        Class A
            ' class code
            End Class
        End Namespace
    NameSpace NS2
        Class B
            ' class code
            End Class
            ' other classes maybe added
        End NameSpace
    End Namespace
End Namespace
```

Namespaces are used for most built-in classes, and the programmer is encouraged to use the **Imports** statement to use the shorthand version of the class names and methods of these built-in classes. For example, in the System.IO namespace, there are many classes for reading and writing to text files, such as the StreamReader class. The two examples below show the short then the long version of the same code. Note that all built-in classes are under the System namespace or a namespace that is nested under the System namespace.

```
' Short version
Imports System.IO ' this precedes any other line including the Class Name
Dim SR As New StreamReader

' Long version
Dim SR2 As New System.IO.StreamReader
```

### 6.7.3 Root Namespace

All namespace names in your project are based or nested under a namespace referred to as the root, which Visual Studio assigns based on the name of the Project. For example, if your project is named Payroll, its programming elements belong to the Payroll namespace. If you declare the Accounting Namespace in that project, the full name of that namespace is Payroll.Accounting. Payroll is referred to as the **Root** namespace of your application.
**Note** that when declaring a namespace, you are not defining a data type, and therefore you cannot assign any access modifiers to a namespace.

This completes our OOP coverage so we can build on that knowledge by learning how to handle errors and group objects. In the next section of the book, we will learn topics that are a direct application of OOP principles.

**Exercises**

6.1 Using the Car and GreenCar classes, allow the client code flexibility by overloading every method at least twice using the same number of arguments but different data types.

6.2 Improve the GreenCar gas consumption by changing the drive method in GreenCar. Do this by calling the parent’s Drive then add another statement that “puts back” gas in the tank, so GreenCar uses 4% less gas than Car.

6.3 Add a new event to GreenCar, ServiceLight, which is raised if the mileage reaches these thresholds: 5K, 10K, then 10K increments. For example GreenCar will need service at 5K, 10K, 20K, 30K and every 10K thereafter. Think about where the event should be raised. Would this event require a different definition of Car’s properties and or methods in the GreenCar?

6.4 Add another property, Serviced which the client can set to False only while the class code can set it True (this requires some thinking about the code itself with a small trick). When Serviced is set to True from the class code, make sure to raise the ServiceLight event under the right condition.

6.5 Organize the classes of your project by creating a new Class Library project and add a namespace called Automobile, and enclose the Car and GreenCar classes in it. Do not add any forms to this project. Add another Windows project that includes the form and its controls. What changes do you need so that you can use the classes of the Automobile namespace?

6.6 Adjust the client code by adding a checkbox that reflects the service needs of the GreenCar objects.
Chapter 7 Exception Handling

This chapter is a prelude to the next chapter. We need to learn all about Exception Handling details before we can move on to chapter 8. The experienced reader is encouraged to complete this chapter to ensure proper and adequate coverage of the prerequisite material required for chapter 8.

Since you have already programmed in VB, then you must be using Try-catch statements to incorporate Exception Handling (EH) in your code. However to get a better grasp on EH you need to learn how and where to handle exceptions, issue or Throw them, and what to do in case you need to act on or ignore them.

You will notice in some literature, references to errors as opposed to exceptions. The actual wording is not the issue, but the construct is. In VB 6.0, Exceptions and the Exception class did not exist, and therefore the concept of Exception Handling did not exist. Instead, it was called error handling, which is a different approach to handling runtime errors that uses GOTO statements and the ERR object. We will not cover error handling since this is an old technique that is not used anymore. Exception Handling on the other hand, is a structured approach. Although error handling is still supported by VB .NET, Exception Handling will be our focus.

7.1 Structured Exception Handling

As a programmer you have learned how to deal with three types of errors: logical, and syntax. The Third type of error is runtime errors which happen when the code executes and tries to do a certain task, e.g. open a file, or get input from the user, but then fails due to 'forces' outside the control of your code. For example, if you have the following:

```vbnet
Dim x As Integer
x = CINT(Text1.Text) ' or x = Text1.Text
```

I can think of numerous situations that may cause the above code to get into a runtime error and halt execution. For example, the user does not type a number, or the user types a number too big to fit into an integer. As a programmer, you cannot blame the user for such errors. Although you can prevent them from happening using different methods such as If statements to check for numeric entries, or to prevent the alpha keys from working while inside the textbox, but there may be other cases, such as reading from failed disk, where such errors may not be prevented since your code can not check for a failed disk. The solution is to add Structured Exception Handling code, or Exception Handling for short, using Try-Catch statements.

The following code shows the structure of a Try...Catch...Finally statement:

```vbnet
Dim x As Integer

Try ' starts an exception handler
    ' code in this section is the code you write normally
    x = CINT(Textbox1.Text) ' place statements that may generate exceptions
    ' Next is the Catch Block where you write code to "trap" exceptions
    ' You may use Catch as a statement on its own or add "filters" for more 
    ' actions based on the type of exception that occurred
    ' 'To add filters, you add a Catch statement for each filter
    'Catch Exception As Exception
    ' 'This is the Catch block where you put code to "catch" exceptions
    ' Next is the Finally Block where you write code to cleanup
    ' Finally code is executed whether an exception occurred or not
    Finally
    ' code in this section is cleanup code
End Try
```

Chapter 7 Exception Handling
'The square brackets below indicate an optional clause

Catch [optional filters: VarName As ExceptionClass]  
Messagebox.Show("error") ' This code runs if the statements listed in 
' the Try block fail and the filter in the Catch statement if used is true.

Finally 'this section is optional altogether  
' Code here executes regardless of whether exceptions did or did not occur  
' and regardless of what the exceptions were  
' This code always runs immediately before  
' the Try statement exits.
End Try ' Ends a structured exception handler.

The Try block of a Try statement contains the section of the code that is expected to cause errors. If an error occurs during the execution of the code in between the Try and the first Catch statement, an object of a Type that inherits from a class named Exception is generated by the compiler. Visual Basic tests each Catch statement within the Try-Catch until it finds a Catch with a Type (or condition) that matches the Type of the current Exception-derived object. The Catch statement is very similar to a Select-Case or If statement. The next section illustrates.

7.2 Catch Illustrated

Catch statements come in two major forms: without a filter, in which case one Catch statement is used to handle any and all exceptions that may occur, or with a filter, in which case multiple Catch statements are used in order to take different action for different exceptions. In that case, care must be used in how to place these Catch statements as their order does matter. The following examples illustrate the difference between using a Catch statement with and without filters.

Try  
'VB Statements  
Catch 'no filter is used  
   Messagebox.Show("An error occurred")  
End Try

The above code, in which the Catch has no filter, has the advantage that any exception that may occur due to the statements in the Try block, marked VB Statements in the above code, can be handled by the one Catch statement shown. The disadvantage, though, is that the error message that will be shown by the Messagebox.show statement can only display a generic message, since we don't know the cause of the error so we can't display any specific information or take any particular action. This may not be an accurate statement, but for now, we will assume it true. To filter for different errors, the solution is to use filtered Catch statements as the next example shows.

Try  
'VB Statements  
Catch a As ExceptionType1  
   'action for cases when ExceptionType1 occurs  
Catch b As ExceptionType2  
   'action for cases when ExceptionType2 occurs  
   ' ... list as many filtered Catch as you think necessary
Catch E1 As Exception 'you may also use an unfiltered Catch here 'generic action when any other exception that those listed above occurs.

**Note** that you may list as many Catch statements for each Exception type that you expect to happen due to the VB statements used. These specifically filtered Catch statements must be placed above the generic or non-filtered Catch statement.

In the above code, assume that VB statements cause an Exception called ExceptionType2 to be thrown – an example is **InvalidCastException**. The flow of code then transfers to the first line of code in the Catch block. The compiler then tries to match the current exception type that occurred, ExceptionType2 to the first Catch type. Since they do not match, control moves to the next Catch statement, in a top to bottom fashion much like If-ElseIf statements work. When the compiler matches the current exception to the one listed in a Catch statement, the exception is said to be handled, and if there are statements within that Catch they are executed. Once an exception is handled and statements within the matching Catch statement execute, control then moves to the Finally block if written, or to End Try.

The code in the Finally section -whether there was an error or not- executes last, so place cleanup code, such as that for closing files and releasing objects, in the Finally section.

Exception handling is about matching the current exception to a Catch statement; the rest is just being user-friendly, so display appropriate and easy to understand messages, and write code to clean up any needed controls or variables. If no Catch that matches the current exception is found, and no non-filtered Catch is used, then the exception is thrown back to the calling code. If the calling code contains a Try-Catch block then the process of matching repeats until the main call is reached. At that point if no Try-Catch block is found that matches the current exception, the exception is said to be unhandled and a runtime error message generated by the compiler is shown and the code execution halts.

### 7.3 What is an Exception?

The birth of an exception can be attributed to two causes. First, are the actions by the runtime caused by the user making certain entries, or failure of code to execute. In that case the compiler looks for a Catch block if the code that caused the exception was within a Try-Catch block. The generation of the exception is nothing but the compiler creating a new instance of an object that inherits from the generic built-in Exception class. Hence, an exception is created when a constructor generates a new instance of an Exception, or an Exception-based class.

The other method through which an exception is created is by the execution of a Throw statement. You may use Throw statements to force the instantiation of an Exception or Exception-based object. This may be needed in cases where a logical check will not cause a runtime error, for example, the user entering negative numbers, or strings of a certain length where the logic requires positive numbers or strings of a minimum length, as in the following code:

```
IF DataEntryVar < 0 Then
   Throw New Exception
Else ....
```

In the above case, we are generating an exception of type Exception. However, we can also generate any exception that inherits from Exception, such as InvalidCastException as in the following code:
If DataEntryVar < 0 Then
    Throw New InvalidCastException
Else ....

Remember that when an exception or an exception-based object is generated, then the compiler will look for a Try-Catch block to handle it, regardless of the source of that exception. We will see how this is an important feature in designing error-proof classes in the next chapter.

### 7.4 The Exception Class

Whenever an exception is thrown, whether through the compiler or through explicit VB statements throwing the exception, a new instance of the Exception class or any of its children is created. The Exception class has properties that aid in identifying the code location, type, and cause of the current exception. For example, the Message property returns a text message describing the error; you can alter it to make a cryptic message easier to understand. You can use this property to set the error message during the throwing of an exception; hence you can assign it a value as in the following example:

```vbnet
Sub EvaluateEntry(DataEntryVar)
    If DataEntryVar < 0 Then
        Throw New Exception("Data entered must be positive only")
    Else ...

View the Throw statement as a call to the constructor of the Exception class. This call passes a String value which is then assigned to the Message property. In the above example, the handling code, which can be placed in the calling code, can then use the Message property to display that particular error message; hence the catch statement can then use the Message property, to display the passed String in a message as in the following code:

```vbnet
'Declare variable X
Try
    'get data into variable X
    EvaluateEntry(X)
Catch E As Exception
    MessageBox.Show(E.Message)
End Try

Note that the code above demonstrates an important concept and that is, when an exception is thrown, control moves to an enclosing Catch statement. Then when the current exception matches the exception type listed in a Catch statement, the current exception is assigned to the variable listed in the matching Catch statement.

In the code above, Exception is thrown during the call, and since a Try encloses the call, the compiler will look for a Catch down the line. When a Catch statement matches the currently thrown exception, which is Exception, then variable E will be assigned to the currently thrown exception, and we can use the Message property of E variable to display it in a messagebox.

Note that if you do not set an error message string in the Throw statement, then the default, which is the one you see during normal execution when no Try statements are applied, is used. Typically the generic exception messages are not meant for the end-user but for the programmer.
Another property of the `Exception` class is `HelpLink` that gets or sets a link to an associated help file. The `Source` property gets or sets a String containing the name of the object causing the error or the name of the assembly – the executable file where the exception originated. We will study the `Exception` class in more details in the next chapter.

**Note** that you should avoid writing code that tests for specific error messages in the Catch block, since messages can be changed by the code that originally produced them. Instead, test for the `Exception` type, such as `InvalidCastException`, or `IndexOutOfRangeException`, etc. In the next chapter we will list more classes that are derived from the base `Exception` class.

### 7.5 Sequence of Events

When an exception occurs in a `Try` block, the system searches the associated `Catch` blocks in the order that they are listed, until it locates a `Catch` statement that matches the exception, otherwise the exception is said to be unhandled and that causes halting of the runtime.

**Recall** from chapter 5 that a child class matches (as in conversion from child to parent succeeds) its parent but not the other way. So based on the examples we did in the previous chapters, if Employee is the parent of Faculty, then an object of type Faculty will match or convert to an object of type Employee, however, the Employee object cannot be matched with the Faculty object. It makes sense that the specialized object (Faculty) can be generalized into the parent (Employee) but the general object (Employee) cannot be specialized through conversion or matching.

Following the above concept and knowing that the `InvalidCastException` is a child of the `Exception` class, then it follows that if an instance of `InvalidCastException` is thrown, that it will match a `Catch` block that uses `Exception` as its filter.

In general, when an exception is thrown, the compiler looks for a `Catch` that filters for that type of exception or its parent. Knowing that child classes match the parent class but not the other way around tells you that a `Catch` statement that filters for a certain type must be specified before a `Catch` statement that handles its base types. In addition, a `Catch` statement that filters for `Exception` must be specified last. Here is an example:

```vbnet
Try
'Code
Catch a As E1
Catch b As E2
Catch c As E3
End Try
```

We omitted the action code usually placed after a Catch to focus our attention on the `Catch` order.

In the code above, assume that the current exception thrown is called EX, and that E1, E2, E3, and EX are all derived from Exception, and are all at the same level in the inheritance hierarchy. Hence EX will not match any of the `Catches` above, and a generic `Catch` is needed after the last `Catch` listed. However, if EX is a child of say E2, then E1 will not match, but E2 will match. For this reason, you should list the generic `Catch` after all the child exceptions are listed. Accordingly, the following code is useless:
Chapter 7

Exception Handling

Try
 'Code
 Catch a As Exception
 Catch b As E2
 Catch c As E3
 End Try

The above code filters for the generic Exception at the top. Since all exceptions that may be thrown are children of the Exception class, then any exception that may occur will match the first Catch statement, making the next 2 Catch statements useless. Remember that we add Try-Catch statements for two purposes:

1. To prevent the breaking of code due to unhandled exceptions, and
2. To display user-friendly error messages, and allow the user to continue and or avoid producing error situations in the future.

Since presenting user-friendly and clear error messages is important, then filtering for the current error, based on which, the error message is produced is an important part in EH code. Hence the order in which the Catch statements are listed is important. Further action may be needed based on the type of the error, hence filtering for types is another reason for properly placing Catch filters.

7.5.1 What Happens After Throw

A Throw statement acts like a Return statement, in that it transfers control out of the block it is in. In the following code, any statements placed after the Throw statement, will not execute.

If a < 0 Then
   Throw New Exception("Negative numbers not allowed")
   a = 0  'this line will never execute
End If
 'this line may execute if a >= 0

7.6 The Call Stack

To get a better understanding of how handling works, we need to learn how code executes during runtime. Assuming you have created a Windows application, and added a module with Sub Main, and that you have identified Sub Main as the startup object. Later in this book we will cover startup objects, but for now, consider if Sub Main looks as follows:

Sub Main()
   Call Sub1
      Statements1
End Sub
Sub Sub1()
   Call Sub2
      Statements2
End Sub
Sub Sub2()
   Messagebox.Show("here")
End Sub
When you click the start button on the application, the compiler calls Main, which in turn calls Sub1, and that causes a call to Sub2, then when we look at Sub2, we see that it will execute the MessageBox.Show statement. Once the MessageBox statement completes, the compiler knows that it must go back to Sub1 and execute the lines that follow the call to Sub2, marked by statements2. Once these statements complete execution, the compiler goes back to Main, and executes the lines that follow the call to Sub1, namely Statements1. But how does the compiler keep track of where to go back to? It uses a list which includes the sequence of the calls, and once it completes a statement, it removes that statement from the list and moves on to the next one to execute it. This list will look something like the following based on the code we have above:

```
Call Sub1
Call Sub2
MessageBox.show("here")
Statements2
Statements1
```

In fact, you can view this line during the runtime using the following steps:

1. Place a breakpoint in the code next to the MessageBox line.
2. Run the code, then when the compiler breaks go back to the IDE.
3. Click the Debug then Windows menu then choose the 'Call Stack' towards the middle of the menu. You will then see the Call Stack window appear.

How does this affect EH code? It affects it because when you use Try-Catch statements, then it does not matter whether the Try-Catch statements are in the current procedure, or in a previous one up the call stack. When an exception occurs, the compiler looks for a Try-Catch statement in the call stack, if one is found and that one happens to be in the current procedure and it contains a Catch that matches the current exception, then we say the exception has been handled. If the current procedure does not contain a Try-Catch, or none of its Catch statements, match the current exception, then the compiler goes to the call stack and searches for another Try-Catch block, and tries to locate a Catch that can match the current exception. If the compiler searches the entire stack and nothing is found, then the exception is thrown to the runtime and the execution halts.

We have added Try-Catch statements to the code above, and changed Sub2 to see how EH in the calling hierarchy works.

```
Sub Main( )
    Try                   'added Try-Catch to Sub Main
        Call Sub1
        'statements1
        Catch e1 As Exception
            MessageBox.show(e1.Message & " but handled in Sub Main")
    End Sub
Sub Sub1()
    Call Sub2
    'statements2
End Sub
Sub Sub2()
    Throw New Exception("Exception thrown in Sub2")
End Sub
```
The code above will create the following calling list:

Try
  Call Sub1
  Call Sub2
  Throw New Exception("Exception thrown in Sub2")
  statements2
  statements1
  Catch e1 As Exception

Hence, when the Throw statement forces the exception to occur, the compiler looks down the list to locate a \texttt{Catch} statement to match the currently thrown exception. As far as the execution is concerned, it does not matter where that matching \texttt{Catch} came from – whether from the current Sub, or the calling one. The compiler simply looks in the Call Stack. What matters is where we go from here – here being the location where the matching \texttt{Catch} was found.

If a \texttt{Catch} that matches the currently thrown exception is found, the exception is said to be handled, and then the compiler executes any statements that follow that \texttt{Catch}, if any are written, then moves to the Finally block, if written, then to End Try, and to End Sub.

Consider another example, where in code that follows, Main calls Sub1.

\begin{verbatim}
Sub Main( )
  Call Sub1
  Call Sub2
End Sub
\end{verbatim}

\textbf{Note} the following important concepts:

1. If Sub1 contains a \texttt{Try-Catch} block, while Sub2 does not contain any EH code, then if an exception is raised in Sub2, then that exception will not be handled by the \texttt{Try-Catch} block of Sub1. The reason is that by the time the code under Sub1 completes execution, then all of its statements would be removed from the call stack, and that includes the \texttt{Try-Catch} block of Sub1. If and when an exception occurs due to Sub2, and as shown, Sub Main contains no EH code, then those exceptions will be unhandled.

2. If Sub1 causes an exception, and that exception matches a \texttt{Catch} statement in the Try-Catch block in Sub1, then that matching \texttt{Catch} statement will execute the statements that follow it. If Sub1, however, does not contain any matching \texttt{Catch}, then the compiler will look for a matching \texttt{Catch} under the caller, Sub Main. If one is found, then its code will execute, otherwise the exception is unhandled.

One thing you will consider doing for the code above, assuming that none of Sub Main, Sub1 or Sub2 have any \texttt{Try-Catch} statements, is to add some. In that case you one many options.

1. Add \texttt{Try-Catch} blocks to Sub1, and Sub2, where some of these \texttt{Catch} statements may be filtering for the same exceptions as shown below:

\begin{verbatim}
Sub Sub1()
  Try
    'code
  Catch a As InvalidCastException
\end{verbatim}
'message code
Catch b As Exception
'message code
End Try
End Sub

Sub Sub2()
Try
'different code
Catch a As InvalidCastException 'same catches as above
'message code
Catch b As Exception
'message code
End Try
End Sub

But if you were thinking the code above is redundant, you are right. Instead of the code above, we have a better option as shown next.

2. To improve the performance in the above code, add a Try block to Main, but remove all Try-Catch blocks from Sub1 and Sub2:

Sub Main()
Try
 Call Sub1
 Call Sub2
 Catch a As InvalidCastException
 'message code
 Catch b As Exception
 'message code
End Try
End Sub

This way, if Sub1 causes say an InvalidCastException, and no Catch statements are written in Sub1, the system searches the previous levels in the current call. The same will happen for Sub2 statements. The difference between the first and second options is that once the exception is handled the execution will proceed to different points in the application.

The approach of Option 2 above is called Centralized EH, and may be used alongside the code of approach 1, whereby Try-Catch statements are also added to procedures in the application. The more centralized the EH is, the more efficient the code is, but the less control it provides to restart the process casing the error. In approach 1, if Sub1 causes an error, execution of Sub2 statements will still proceed. The less centralized the EH code, referred to as Distributed EH, the more control it provides for the programmer, and eventually the end-user, to restart from the prior point to when the error occurred. Since our focus in this book is the OOP approach to EH, the interested reader should search the web for more resources on centralized vs. distributed EH design.

With the above coverage, we are now ready to learn how to create our own exception classes, called User-Defined-Exceptions, as the next chapter details.

Exercises
7.1 In the previously designed Car and GreenCar classes, there were situations where the code checks for logical errors. For example, Gas level must be positive, Vin Number must be alphanumeric only, or Miles driven must be positive.

1. Add code to check for these logical errors, so that when an error occurs, the class code would issue an error message via MessageBox.
2. Now contrast that with EH and replace every MessageBox showing an error from Car or GreenCar with a Throw New Exception statement. Generate an error message in the Throw statement that describes the error.
3. Step 2 above, will require that the client code handle these exceptions using Try-Catch statement. Add Try-Catch statement where the code is expected to cause exception in the client code.
Chapter 8 User-defined Exceptions

In the previous chapter you learned how to handle exceptions and throw them, but these were all the built-in exceptions that the VB language defines. You also learned that all these exceptions were derived from the Exception base class. In this chapter you will learn how to create your own Exception-based classes, so you can throw your own exceptions called user-defined exceptions. Before we learn all the details about user-defined exceptions, we need to learn more about the parent, the Exception class.

8.1 The Exception Class

The Exception class includes a number of properties that help identify the code location where the exception occurred, the type, the help file, and the reason for the exception. Some of these properties are:

- **Message** is a String property that contains human-readable text that describes the error. When an exception occurs, the runtime makes available a text message to inform the user of the nature of the error.

- The **StackTrace** property contains information that can be used to determine where in the code the error occurred. The stack trace lists all the called methods, and the line numbers in the source code where the calls were made. You can use this property to locate “bugs” in your code that can be cleaned up in newer versions. You can also use this information to have the end-user send a report to the software developer to fix future versions.

- To provide the user with extensive information concerning why the exception occurred, the **HelpLink** property can hold a URL to a help file. The programmer needs to create this file and load it onto the website referenced in this property.

- The **Source** property helps determine the name of the application or the object that caused the error.

The Exception class has also the *InnerException*, *HResult* and *TargetSite* properties, but they are not the focus in this chapter. We will work with the other four properties listed above in the next section.

8.2 Exception Hierarchy

There are two types of exceptions: exceptions generated by an executing program which inherit from *ApplicationException* and exceptions generated by the common language runtime which inherit from *SystemException*. The Exception class is still the base class for all exceptions.

Most exceptions are derived from either *SystemException* or *ApplicationException*. Mostly, all the runtime generated exceptions inherit from the *SystemException*. For example, the *InvalidCastException* Class hierarchy is as follows, where Object is the top parent:

```
Object
   Exception
```
SystemException
InvalidCastException

The runtime throws the appropriate derived class of SystemException when errors occur. These errors result from failed runtime checks (such as array out-of-bound errors), and can occur during the execution of any method.

On the other hand, an ApplicationException is thrown by a user-program rather than by the runtime. Remember you can **throw** a system defined exception, or you can create your own class that inherits from the Exception class or from the ApplicationException class, then throw those exceptions.

If you are designing an application that creates new exceptions, you should derive those exceptions from the ApplicationException class. It is not recommended that you throw a SystemException in your application. View System Exceptions as errors that violate the compiler's rules, while Application Exceptions as ones that violate your logic.

In short, all the system-defined exceptions derive from the SystemException class, which you should not inherit from. The ApplicationException class derives from the Exception class, and all the user-defined exception classes that you need to define should inherit from the ApplicationException class.

8.3 Using User-Defined Exceptions

If you want users to be able to programmatically distinguish between some error conditions, or if you want to design a class that forces out any illegal assignments of properties, you can either **throw** exceptions of type Exception, or create your own user-defined exceptions and **throw** them. During the creation of the exception object, you can pass a text String to the constructor to describe the details of that particular exception. If no error message argument is supplied to the constructor, the default error message is used. The next section provides more details.

The following lists the steps needed to create and use User-Defined Exceptions (UDE):

1. Write code to create the UDE which inherits from the Exception or the ApplicationException classes. When creating your own exceptions, it is good coding practice to end the class name with the word "Exception", as in MyOwnException.

2. Even though you are only required to develop one constructor to call the parent constructor, it is a good practice to implement the two class constructors shown below, which define a new class, EmployeeListNotFoundException, that is derived from System.ApplicationException. Two constructors are defined each taking different parameters. The difference between the two versions is the string message that is generated and passed on to the handling code.

   ```plaintext
   Public Class EmployeeListNotFoundException
   Inherits ApplicationException
   Public Sub New()
   'constructor 1
   MyBase.New()
   'more details on this below
   End Sub

   Public Sub New(message As String)
   'constructor 2
   'the message that is created during the Throw statement will be passed
   ```
'into the message argument then to the Exception parent class into 
'the Message property in the next statement 
MyBase.New(message)
End Sub 'New

End Class 'EmployeeListNotFoundException

In constructor 1, the parent constructor is called without arguments. This will cause the built-in generic message to be used as the Message property. An alternative to this argumentless call is a call with a string message such as:

MyBase.New("Error: Employee list not found")

In the code above, the string is saved into the Message property of the parent class. This argumentless constructor is used when the instantiating code looks like the following:

Throw New EmployeeListNotFoundException

In constructor 2, the parent constructor is called with a String message. This message is passed on from the Throw statement. For example:

Throw New EmployeeListNotFoundException("Error in code")

In the Throw statement above, a new instance of the exception-derived class is created by calling constructor 2, which receives the string "Error in code" and passes it to the Message property.

3. In the class code that does logical checks, **Throw** the UDE when needed. This may be in Employee Class code. Note the UDE class must be either added as a class to the project using Project, Add Existing or Add New, or a reference must be added to it, using Project then Add Reference if the class is compiled.

For example, the following code would be part of the Employee class, or Faculty class, and it needs the exception class to be added or referenced:

```vbscript
Sub SomeSub()
    If SomeCondition Then
        Throw New EmployeeListNotFoundException
    ...
End Sub
```

4. In the code you are handling, catch the UDE or the Exception class. This code would typically be on the user code, or the form where an object based on the Employee class is used.

```vbscript
Try
    Call SomeSub
    'OtherStatements
Catch a1 As Exception 'or EmployeeListNotFoundException
    Messagebox.show(a1.Message)
EndTry
```
Recall that EmployeeListNotFoundException is derived from the built-in ApplicationException class which in turn is derived from Exception. This means that the UDE thrown will match the Exception class. Hence in the Catch above, the messagebox statement will show whatever error message that was generated by the Throw statement or the constructor code.

Note that a general misconception to add Try-Catch statement to the same code that throws the exception-based object does not add any value. The reason is that we are using Throw statements in the Employee class to replace the previously MessageBox statements which violate the privacy of source-client code communication, and assume that the client is a Windows application. Adding Try-Catch in the employee class does not communicate the error to the client code nor does it allow the client code to inform the end-user of the error situation.

8.4 Exception Handling Best Practices

A well-designed exception handling code can make a program more robust and less prone to crashing because the application handles such errors. In this section we will discuss some suggestions on best practices for exceptions throwing and handling.

Know when to set up a Try/Catch block. For example, you can programmatically check for a condition that is likely to occur without using exception handling. In other situations, using exception handling to catch an error condition is appropriate. For example, the code below uses an If statement to check whether an entry is numeric.

```vbnet
If IsNumeric(Textbox1.Text) Then
    If CInt(Textbox1.Text) > 0 Then
        ...
    End If
End If
```

The same logical condition above is dealt with using Try-Catch statements instead in the code the follows. If the not numeric the source code will throw an exception, which is handled by the Catch statement.

```vbnet
Try
    If CInt(Textbox1.Text) > 0 Then 'if entry is not Numeric CInt throws error
        ...
    End If
Catch ex As Exception 'the thrown exception will be handled here
    'action in case of error
End Try
```

The method you choose depends on how often you expect the situation to occur. If the event is truly exceptional and is an error (such as an unexpected end-of-file), using exception handling is better because less code is executed in the normal case. If the event happens routinely, using the programmatic method such as If statements, to check for errors is better. In this case, if an exception occurs, the exception will take longer to handle.

Use Try/Catch blocks around code that can potentially generate an exception and centralize your Catch statements in one location. This way, the Try statement potentially generates the exception, the Catch statement handles the exception from a central location and
the *Finally* statement closes or de-allocates resources. We have covered centralized EH in the previous chapter.

**Always order Exceptions in Catch statements from the most specific (newest child) to the least specific (oldest parent).** This technique handles the specific exception before it is passed to a more general *Catch* filter. The technical details are related to inheritance and that child class types match the parent class type. For more details refer to chapters 5 and 7.

**End exception-derived class names with the word "Exception".** This is only a naming convention that Microsoft uses.

**In most cases, use the predefined exception types.** Define new exception types only for programmatic scenarios. Introduce a new exception class to enable a programmer to take a different action in code based on the exception class. For example if the action taken when *InvalidCastException* occurs must be different from when EmployeeNotFound occurs, then do create and throw the latter exception. Otherwise, just throw the *InvalidCastException*. There is no need for a new type.

**Do not derive user-defined exceptions from the Exception base class or the SystemException class.** For most applications, derive custom exceptions from the *ApplicationException* class.

**Include a localized description string for every thrown exception.** When throwing exceptions add a descriptive message, so this message is passed on properly to the client code. Ensure that the UDE class uses a constructor that has at least the *Message* argument as explained in step 2 of the Steps to create UDE section earlier in this chapter. Since all derived exceptions will match *Exception* in a *Catch* filter, this means the Try-Catch statements do not need to use many Catch filters to display custom error messages. In that regard, use grammatically correct error messages, including ending punctuation. Each sentence in a description string of an exception should end with a period.

**Design classes so that an exception is never thrown in normal use.** For example, a *StreamReader* class exposes another way of determining whether the end of the file has been reached. This avoids the exception that is thrown if you read past the end of the file. A *StreamReader* object uses the *Peek* method which returns -1 if the end of file is reached. Checking whether the *Peek* function returns -1 replaces the need for *Try-Catch* blocks.

**Throw an InvalidOperationException if a property Set or method call is not appropriate given the object's current state.** For example if an object has not been created, yet code is attempting to set one of its properties.

**Throw exceptions instead of returning an error code.** For example, a function returns the average of an array, but if one of the array values is negative and that constitutes a logical error then in that case the function should *Throw* an exception instead of returning the average without counting the negative value, or simply returning -1.

**Clean up intermediate results when throwing an exception.** Callers should be able to assume that there are no side effects when an exception is thrown from a method. For example, if a new value is assigned to a property, and the Set code detects an invalid value for the property, before throwing the exception, make sure the old value stays as is by not assigning the new value at all. The following code shows an example:
Set (ByVal Value As Type)
    If Value < 0 Then
        'no code should assign Value
        Throw New SomeException
        'any code here will not execute
    End Set

**Exercises**

8.1 In the previous chapter exercises the Car and GreenCar classes were updated so that when logical conditions were violated, exceptions were thrown. Add new ApplicationException-derived classes that will be thrown instead.

8.2 How does the use of user-defined exceptions change the Try-Catch blocks on the client-side code? Can you code the UDE and throw them so that the client code does not need to add more Catch statements?

8.3 To improve the performance of the client code, there may be cases where If statements may replace Try-Catch blocks, can you locate at least one such case, and why checks are better than Catches?

8.4 How would you reduce Try-Catch blocks and use more centralized EH code?

8.5 What would a good situation be where the Car and GreenCar classes would return values indicating errors vs. throwing exceptions?
Chapter 9 Collections

Managing multiple items of the same type is what this chapter and the next are all about. The concept should not be strange since you have already been using arrays, which allow you to group data of the same type. If you have been using listboxes and comboboxes, then you should also be familiar with the concept. In the next chapter we will cover more on arrays, but for now, we will look deeper into a new technique to manage objects of the same type into a collection.

A collection is a construct, like an array, that allows you to manage objects of one type. Recall in previous chapters, you designed and wrote code for the Employee class. On the client code, you were able to create new Employee objects by instantiating an object based on the Employee type as in:

Dim Clerk As New Employee

Then later you had code, also on the client side, which manipulated the properties and responded to events of Clerk. The only problem was that each time the above statement executed, the previous Employee object, Clerk, with all its properties values was 'dumped' into Garbage, i.e. lost in RAM. The newly created Clerk object started off with no set properties values other than the defaults. But does this mean that the client code can only work with one Employee at a time, and once a new one gets created, the old one is thrown out? The answer has to be no.

9.1 Collections vs. Arrays of Objects

One way to manage multiple instances of a type, like Employee is to use an array of an object type just as you would declare and use an array of any data type. The members of this array can be retrieved by their index, which starts at 0, and can be manipulated as any object of this type would be. Arrays also have built-in functionality for searching and sorting that can be accessed through the array variable. To create an array of objects, follow the steps below.

1. Declare the array as shown in the sample code below. Because arrays are zero-based, they contain one member more than the upper bound you declare them with.

   Dim x(10) As Employee ' Contains 11 members, from x(0) to x(10).

2. Instantiate each member of the array, or assign each member a reference to an already existing object. An example of each approach is shown below:

   ' Option1: Instantiate each member of an array using a loop.
   Dim q As Integer
   For q = 0 To 10
      x(q) = New Employee() ' code later will assign property values
   Next q

   ' Option2: assign a member of an array a reference to an existing object.
   Dim Clerk As New Employee()
   x(0) = Clerk

   Note that you can assign the different members of the array references to the same object. Hence, x(1), x(2), etc can all point to Clerk above.
In the case of the second option, you can use:

```csharp
x(0).FirstName = "John"
x(0).Salary = 45,000
```

As so on, but the above code indicates that the burden on managing all the employees in the array `x` is on the programmer of the client code. For example, if we need to know what the average Salary is for that array, that code, which would be a loop that adds up all the Salary values from `x(0)` to the array's upper bound then divides that total by the number of entries in the array, would be written by the programmer on the client end. It would provide more functionality and convenience if we create different functions that work on groups of employees, and the end user code programmer needs only to invoke that code, but not write it. This is possible through collections. The other disadvantage of arrays is that if you need to add more objects than its current `UpperBound`, you will have to `Redim` the array to a newer size. As we cover more topics in this chapter you will also see other disadvantages that arrays may produce.

A collection on the other hand, is similar to an array in that it allows you to group items together, however collections provide more flexibility, functionality and power. A collection is a class that you can create, and indicate through the code that objects in this collection can be of a certain type, or any other types that derive from it. Hence you can create a collection of `Employee`, but add to it objects of type `Employee`, `Faculty`, and any type that inherits from `Employee`. Further, on the client side, you don't need to specify how many objects you plan to add to the collection, making the code more compact.

There are two major ways to create a collection class in VB, the first involves creating a class that inherits the base class called `CollectionBase`, then creating an object based on that class; the next section provides details. The other way is to use VB’s built-in collections such as the `Collection` class and create an object of the `Collection` type, and add the `Employee` objects to it. However, this method provides less functionality, and objects of any type, not just `Employee` can be added to the collection.

In addition to the `Collection` class, the .NET framework provides several collection-type classes in the `System.Collections` namespace. Some, such as `Stack`, `Queue`, and `Dictionary`, are specialized classes that have been implemented to fulfill specific roles. We will cover `Stack` and `Queue` in the next chapter, but will skip `Dictionary` to focus on the main subject of grouping objects, and providing only the coverage needed to review the alternatives.

The other classes such as `CollectionBase` and `DictionaryBase`, are `MustInherit (abstract)` classes that have some basic functionality in place, but leave much of the implementation to the developer. Again, we will focus on the `CollectionBase` class. Recall that a `MustInherit` class is not a type, and must be inherited in a derived class.

**9.2 Creating CollectionBase Classes**

You can create your own collection classes by inheriting from `CollectionBase`, which already has implementations for a `Clear` method and a `Count` property, and it maintains a `Protected` property called `List`, which it uses for the internal storage and organization of the objects that it contains. Other methods, such as `Add` and `Remove`, as well as the `Item` property, require implementation.

The basic idea here is that the `List` property has its own Add and RemoveAt methods and the `Item` property already implemented, and the developer can use those to create their own
"wrapper" methods based on the List's **Add**, **RemoveAt** and **Item** to work on the collection. In other words, the implementation of Add, Remove and Item of your collection, will be simple calls to the List's own methods, which is a simple process.

The example that follows uses the **CollectionBase** class to create a class called EmployeeCollection. This is a collection that accepts only objects of type Employee. You then implement methods to add and remove objects of type Employee to your collection, and you also implement an **Item** property that returns the employee object at a specific location or index.

To create your own **CollectionBase** derived class, ensure that you have already developed the Employee class first. This Employee class was discussed in earlier chapters, so review the definition and create the code accordingly. Below, we have a small version of the Employee class for practice. Use the following only if you have not developed the Employee class code already.

1. Open a new Windows Application and name it **Employees**. From the Project menu, choose **Add Class**. In the Add New Item dialog box, name your class **Employee**. The Code Editor for the Employee class opens.

2. Add the following code or use the previously developed Employee class:

```vbnet
Class Employee
    Dim mFirstName As String
    Dim mLastName As String
    Dim mSalary As Single

    Public Property Salary As Single
        Get
            Return mSalary
        End Get
        Set (ByVal Value As Single)
            If Value > 0 Then
                mSalary = Value
            Else
                Throw New Exception("Salary must be positive")
            End If
        End Set
    End Property

    Public Property FirstName As String
        Get
            Return mFirstName
        End Get
        Set (ByVal Value As String)
            If Value <> Nothing Then
                mFirstName = Value
            End If
        End Set
    End Property

    Public Property LastName As String
        Get
            Return mLastName
        End Get
End Class
```
Set (ByVal Value As String)
    If Value <> Nothing Then
        mName = Value
    End If
End Set
End Property
Public Sub Promote()
    Salary = Salary * 1.07
End Sub
End Class

3. Save the project.

With the Employee class implemented, the next step is to create the collection code through the EmployeeCollection class.

To create the EmployeeCollection class, follow the steps below:

1. From the Project menu, choose Add Class. In the Add New Item dialog box, name your class EmployeeCollection. Note that the name does not need to include the word collection but this a naming convention we will follow.

2. In the code editor, add code to cause the class to inherit from `CollectionBase`, as in this example:

   ```
   Public Class EmployeeCollection
       Inherits System.Collections.CollectionBase
   End Class
   ```

3. Save the project.

Since EmployeeCollection inherits from `CollectionBase`, it already has the following functionality implemented:

- There is a `Clear` method to clear the collection of its content.
- The `Count` property keeps track of the number of current members.
- The `List` object which is a `Protected` object, which child classes can use to keep track of the Employee objects. List has the `Add`, `Clear`, and `RemoveAt` methods already defined. Because List is declared `Protected` in the parent class only child classes have access to it. Moreover, List allows variables of type Object to be added to it, but we want to restrict the type of objects added. In addition, we don't want the client code to directly manipulate List, and the client programmer will not enjoy working with List, hence we must implement code to work on list, but allows the client programmer easy access to functionality, but restricted only to the Employee Type.

To allow for minimum functionality, and restrict types to the Employee class type we must develop at least code for the `Add` method, the `Remove` method, and the `Item` property. Other functionality may also be added as shown later. A minimal example is shown below.

To implement the Add and Remove methods follow these steps:
1. Add the following code to the EmployeeCollection class, beneath the *Inherits* statement:

```vbc
Public Sub Add(ByVal Emp As Employee)
  ' Invokes Add method of the List object
  List.Add(Emp) 'Recall List is Protected, so child classes have access to it
End Sub

*Note* the collection restricts objects added to type Employee by using the Employee type in the Add argument as shown above. Though the *List* object can accept any type of object, our Add method prohibits any objects but Employee types from being added, and acts as a "wrapper" around the *List's* Add method.

2. Save the project. Next, we code a Remove method. Keep in mind that the *List*, has a *RemoveAt* method which removes an object at a specified index which is 0 based. Hence, our Remove method should also pass on an index value. You can name the method Remove or RemoveAt (or any name for that matter).

3. Beneath (or above) the Add method, add the following code:

```vbc
Public Sub Remove(ByVal index as Integer)
  'Index must be a positive integer
  'Recall that Count is a built-in *CollectionBase* defined property
  If index > Count - 1 Or index < 0 Then
    'List is zero based- hence Count-1 is the max possible index
    'If no Employee at that location exists, throw exception
    Throw New Exception("Index not valid!")
  Else
    ' call the RemoveAt method of the List object.
    List.RemoveAt(index)
  End If
End Sub
```

This method accepts an integer value for the index argument. If the value is valid, it is passed to the *RemoveAt* method of the *List* object, thereby removing the item at the indicated index from the collection. The index is invalid if it is less than 0, or greater than the number of items (Count) – 1.

You may also add other checks based on the logic of your application. For example, if the application requires that there is at least one item, then you may check that the count must be at least 2, since after the removal, only one item is left.

4. Save the project and add the *Item* property which allows you to obtain a reference to an object in your collection using its index. Since you already have an *Add* method to add members to the collection, *Item* will be a *ReadOnly* property, though this depends on the logic of the application.

5. Continuing in the EmployeeCollection class, add the following code then save the project:
'Declare the Item property as ReadOnly, and it will return an Employee type
Public ReadOnly Property Item(ByVal index as Integer) As Employee
    Get
        ' You may add code to check that index is valid as in Remove above
        ' Recall List maintains items as Object type, hence we must cast
        ' Cast explicitly to the Employee type, and return to the caller
        Return CType(List.Item(index), Employee)
    End Get
End Property

In the above code note the following important concepts:

- The index must be an integer, and typically should be checked for legal range just as in the Remove method coded previously.
- For proper technique, the return type must be declared as the type that the current collection adds. We have already discussed that derived classes match their parent type. Hence any child of Employee may also be added and returned through item.
- It is also proper technique to cast or convert the type using the CType function. This conversion is needed since List is of the Object type.

Another issue to note is that it is typical in most predefined collections in the .NET framework, that the Item property is the default property, hence you will often see the syntax CollectionObjectName.Item(0) frequently interchangeable with CollectionObjectName(0). If you want your collection to support this syntax, you should make the Item property the Default property.

You have now implemented the basic collection functionality in your EmployeeCollection class. To recap, the EmployeeCollection class now has the following methods and properties defined:

- **Count**, which is defined by the parent **CollectionBase**. Count returns the number of the EmployeeCollection members.
- **Add**, which requires an argument of type Employee. You may add code under this method that checks if the Employee object is legal before adding it to List. These checks could be a unique ID, valid length of a String, such as the social security number. Make sure to **Throw** an exception if the object to be added violates the logical rules.
- **Remove**, which requires an index integer argument. Make sure to add range checking code to ensure that the passed index is legal.
- **Item**, which is a **ReadOnly** property that requires an index and returns an object of type Employee. Again make sure the index is in a legal range, and remember that this property does not need to be **ReadOnly**. Making it as such protects objects from being overwritten in case the client code attempts to add a new Employee in a location that is already occupied.

**Note** that the names of the methods and properties that you define need not be as stated above. For example, you may define the Delete method instead of the Remove method. The choice of the method name is up to you, but remember that programmers prefer and expect the method names used above, since these are the names used in most predefined .NET collections.

**Note** also that based on the application logic, you may need to implement other functionality, such as the average salary, or the highest Rank, or the number of employees in a certain department. In other cases, like Account as a class and AccountReceivable as a collection, the
total receivable balance, or accounts more than 60 days overdue. We will cover some of these examples later in this chapter.

Now that the EmployeeCollection class is complete, it is time to test the functionality. To do this, you need to create a Windows application to the existing Employee and EmployeeCollection classes that will add, remove, and loop through the Employee objects in your collection. To create the test project use the Project menu and add a form if one is not already available and follow the steps below.

1. On the form named Form1 add two Label controls, two TextBox controls, and three Button controls to Form1.
2. Set the properties as follows:

<table>
<thead>
<tr>
<th>Control</th>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label1</td>
<td>Text</td>
<td>&quot;First Name&quot;</td>
</tr>
<tr>
<td>Label2</td>
<td>Text</td>
<td>&quot;Employee Index&quot;</td>
</tr>
<tr>
<td>TextBox1</td>
<td>Text</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>TextBox2</td>
<td>Text</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>Button1</td>
<td>Text</td>
<td>&quot;Add an Employee&quot;</td>
</tr>
<tr>
<td>Button2</td>
<td>Text</td>
<td>&quot;Remove an Employee&quot;</td>
</tr>
<tr>
<td>Button3</td>
<td>Text</td>
<td>&quot;Review Employees&quot;</td>
</tr>
</tbody>
</table>

In the design of your form, Label1 should label TextBox1, and Label2 should Label TextBox2. Change the form name to EmployeeTest.

3. On the code view of the form, type the following code to create a form-level object:

```
Dim myEmployeeCollection as New EmployeeCollection()
```

4. In the designer, double-click button1, and add the button1_Click event handler, using code to add a new Employee to the collection and set the first name of this Employee to the contents of TextBox1.

```
If textBox1.Text = "" Then
    MessageBox.Show("Please enter a first name")
Else
    ' Declare a new Employee
    Dim Clerk as New Employee()
    ' Set the FirstName field
    Clerk.FirstName = TextBox1.Text
    ' Add the new Employee to the collection.
    myEmployeeCollection.Add(Clerk)
    ' Update TextBox2 with the index of the Employee that was added.
    ' The index of a zero-based collection is Count property minus 1.
    TextBox2.Text = (myEmployeeCollection.Count - 1).ToString()
End If
```

5. In the Button2_Click event handler, add code to remove the Employee at the index specified in TextBox2.
Try
    myEmployeeCollection.Remove(CInt(TextBox2.Text))
Catch ex as Exception
    MessageBox.Show(ex.Message)
End Try

Note that there is no need to check if a valid value for index exists in TextBox2, because the EmployeeCollection.Remove method checks this automatically. It is for this reason, that you should add the range checking code in the collection class code, since you cannot rely on the client code to do so to prevent any leaks causing errors.

6. In the Button3_Click event handler, write code to cycle through all members of the collection and display their names in a MessageBox statement.

    Dim counter as Integer
    For counter = 0 to myEmployeeCollection.Count - 1
        MessageBox.Show(myEmployeeCollection.Item(counter).FirstName)
    Next counter

Note that the code above could also be implemented in the collection code, so the client code would only call the function and it would return the String data. The following code would be implemented in the collection code:

    Public Function ShowAll() As String
        Dim i As Integer
        Dim S As String = ""
        For i=0 To (List.Count – 1)
            S = S & List.Item(i).FirstName & ControlChars.CR
        Next
        Return S
    End Function

Then the client code can use:

    MessageBox.show(MyEmployeeCollection.ShowAll)

Finally, an alternative to For-Next loops is a For-Each loop. We will cover it in other examples later in this chapter, but here is the alternative:

    Public Function ShowAll() As String
        Dim x As Employee
        Dim S As String = ""
        For Each x in List
            S = S & x.FirstName & ControlChars.CR
        Next
        Return S
    End Function

You will now run your application and test the Add and Remove methods, and the retrieval of Employee objects from the collection through the Item property. To test the collection, follow the code below:
1. In the FirstName TextBox, type any first name and click the Add an Employee button. Repeat this two times, using different first names. Three Employees have now been added to the collection.

2. Click the Review Employees button. The first name of all employees appears in a MessageBox statement.

3. In the Employee Index textbox, type 1 and click the Remove an Employee button. The Employee at index 1 is removed; note that this is the second Employee. Once the second Employee is removed, the index is updated dynamically, so that the Employee that previously occupied index 2, now occupies index 1.

4. Click the Review Employees button again to check the list of employees now.

In the previous exercises, you learned how to implement basic collection functionality by inheriting from the System.Collections.CollectionBase class. VB has also a built-in Collection class, that is fully defined, and all you need to do to use it is to declare an object based on the Collection class. The next section provides more details.

9.3 Predefined Collections

There are many collections that you have been using through the use of controls such as Listboxes and Comboboxes. In addition there are other collections that the compiler uses implicitly without you seeing the actual code that manages them while you have been creating Windows applications.

For example, every form has a Controls collection. This is an object that represents all the controls on that form. It allows you to obtain a reference to a control in the collection by its index, and to loop through the members of the collection using For-Each statements, without having to list the control name. This allows you to do group actions such as reset common properties, such as the Text property. The code below is an example that resets all textboxes on a form named form1, where Textbox is the class that defines Textbox controls:

```vbnet
Dim x As Textbox
For Each x In Form1.Controls
    x.ResetText()
Next
'note x above can be declared within the For line as in:

For Each x As Textbox In Form1.Controls
```

An Important concept to watch for here, is that the type of object x –set to Textbox- allows the code in the loop to only work on Textbox objects in the Controls collection. So objects in Controls above, that are not of type Textbox, will not be ‘scrolled through’ in the loop body. This makes the code very compact and rids you of having to use If statements to check for Textboxes only in the collection.

In addition to the Controls collection, Visual Basic provides many other predefined collections, one of which is a very general one that a programmer can use called the Collection class. Like a form’s Controls collection, the Collection class also provides you with the built-in functionality to loop through members using For-Each and to reference members by an index. Since both are
collections, why then does the following code from a Windows Forms application generate a compiler error?

Dim myControlsCollection As Collection
myControlsCollection = Form1.Controls ' This generates a compiler error

What's happening here? The Controls collection is a collection; the variable myControlsCollection is declared as Collection; why can't you assign a reference from Controls to the variable myControlsCollection?

The reason is that the Collection class and the Controls collection are not interchangeable, because they are different types with different implementations. They don't have the same methods, nor store object references in the same way, or use the same kinds of index values. The above example shows that not all Collections are created equal, and therefore you should be aware when coding.

9.4 One-Based Collection and Zero-Based CollectionBase

There are many ways to implement user-defined collections. We already covered one of them namely the CollectionBase. The second way is by declaring an object of type Collection as shown in this section. Before we detail how to work with an object of type Collection, it helps to get the overall picture of the difference between creating a Collection object and creating a new class that inherits the CollectionBase class.

A Collection typed object is 1 based and gives you access to its built-in methods and properties of Item, Add, and Remove. The CollectionBase, however, is 0 based and gives you the Clear method, the Count property, and List object with its methods/properties. However it has no Add, Remove or Item. So why bother and write all the code for a derived class from CollectionBase?

The .NET Framework is standardizing collections as being zero-based. The VB Collection class is one-based primarily for compatibility with previous versions. Use this class only when writing code to interact with older versions of Visual Basic prior to the .NET version.

In addition to being one-based, an object of type Collection allows you to add objects of any type – hence Object type, and you can access an item using either a numeric index (Integer Data Type) or a String key. By contrast, some collections (such as System.Collections.ArrayList which we cover in the next chapter) allow only a numeric index. These collections provide access to their members only through the index, and do not allow you to associate a key.

9.5 Using the Collection Object

Since the Collection class is already defined, all you need is to create an instance of an object based on the type Collection. Once the Collection object is created, items can be added, removed, and otherwise manipulated through it. In the example that follows you learn how to create and populate a Collection object.

To create a collection, declare and instantiate a Collection variable as shown in the sample code below:

Dim myCollection As New Collection
Then use the `Add` method to add members to your collection, and remember that these members could be of any type. In this example, you create four strings and add them to your collection. A unique `String` value may optionally be added as the key for each member of the collection. This value is passed to the collection as the second argument of the `Add` method.

```vbnet
Dim w, x, y, z As String
w = "Wow!"
x = "It's"
y = "A"
z = "Collection"
myCollection.Add(w, "This") 'Here we're adding w with a key of "This"
myCollection.Add(x, "Is") 'another object with the "Is" key
myCollection.Add(y, "A")
myCollection.Add(z, "Key")
```

If you want to remove any member of your collection, you can use the `Remove` method based either on the index of the member, or on the optional key. Remember that the index of the first member is 1 not 0. Examples are shown below:

```vbnet
myCollection.Remove(1) 'Removes first member of the collection.
myCollection.Remove("Is") 'Removes the member with the key "Is".
```

**Note** that when a member is removed from a collection object, the index values are renumbered from one to the `Count` value.

**Note** also that you can use the `For-Each` statement to loop through and process the members of your collection, as in the continuation of our example below. To use a For-Each to process members of a collection, declare a variable of the type you want to process in your collection. For example, if you want to loop through all the strings, declare a variable of type `String`, as shown below:

```vbnet
' Code from the previous example goes here.
Dim aString As String
For Each aString In myCollection
    If aString = "Collection" Then
        MessageBox.Show(aString) 'Displays "Collection".
    End If
Next
```

Use the `For-Each` statement to examine each member of your collection. In this example, search for a particular string, and if you find it, display it in a `MessageBox` statement.

```vbnet
For Each aString In myCollection
    If aString = "Collection" Then
        MessageBox.Show(aString) 'Displays "Collection".
    End If
Next
```

### 9.6 Inside a Collection Object

A `Collection` object stores each item as an `Object`; thus the list of things you can add to a `Collection` object is the same as the list of things that can be stored in an object variable. This includes standard data types, objects, and arrays, as well as user-defined class instances. However because the `Collection` object stores each item as an object, the `Item` property's return data type is `Object`. This presents the issue of how to access collection members later. If `Option Strict` is `Off`, you can implicitly convert a reference from the `Collection` to the appropriate type, as shown in this example.
Option Strict Off 'above the Class declaration
Dim myString As String = "This is my String"
Dim aString As String
Dim myCollection As New Collection()
myCollection.Add(myString)
aString = myCollection.Item(1) ' Collection item converted to string.

If Option Strict is On, however, you need to cast from Object to the type the collection contains. To obtain a reference from Item in this way, you can use the CType function to explicitly perform the conversion, as shown below:

    aString = CType(myCollection.Item(1), String)

Now that you learned about Collection and CollectionBase objects, it is time to learn about other ways to group objects such as the use of Stacks, Queues, and Lists. In the next chapter we learn about these constructs in more details.

Exercises

9.1 Using the Car and GreenCar classes, create a new class to keep a collection of Cars and GreenCars with a zero-based index. Make sure the client code can add, remove and point to a particular object. Add the required buttons so the client can choose a Car or GreenCar object, drive it and fill it with gas.

9.2 Add more code to the collection class code, so that the client can get inventory information about the total gallons of gas in the entire collection of Cars and GreenCars. In addition add code so the client can get information such as total mileage on all cars.

9.3 Add another function to the collection so that the client can use a LowestMileageItem as an alternative to the Item function.

9.4 In previous chapters you learned how to create and throw user-defined exceptions. Create a new user-defined exception called DuplicateVINException. This will be thrown by the collection code whenever the client code attempts to create a new car that has a VIN that is already used by another car in the collection. In addition to the new code, what updates are needed, so that when the user tries to add such a car, the error message is shown, and the car is not useable.
Chapter 10 Stacks, Queues, and Lists

This chapter closes on grouping objects, as we cover different type of collections implemented in VB. As you know, the System.Collections is a namespace that contains many interfaces and classes. We will only cover a few of those, namely, ArrayList, SortedList, Stack and Queue. The interested reader is referred to www.msdn.microsoft.com for more information about the other classes and interfaces of the Collections namespace.

The four Collections that will cover differ in how they arrange objects. ArrayList is like arrays, but the size increases automatically as you add more elements and exceed the capacity. The order in which the objects were added determines the order (or index) of these objects. SortedList on the other hand, allows you to manage a collection but each object is referenced through either its index or a unique key. The items in the collection are sorted (hence indexed) based on the key value. A Stack allows you to add objects, where you can access them in a First-In-Last-Out or FILO order, so the first object added has the last index, whereas a Queue allows you to group object in a First-In-First-Out or FIFO order.

10.1 ArrayLists

An ArrayList allows you to group objects like an array does, but you don’t need to resize the ArrayList as it doubles in size if you try to add objects when it exceeds its capacity. The data type that can be added is any type, but the item stored in an ArrayList is officially of the Object type. The capacity of an ArrayList is the number of elements the ArrayList can hold. If RAM management is an issue, then the capacity can be decreased by calling the TrimToSize function or by setting the Capacity property explicitly. The Count property shows how many elements are there in the collection. Elements in this collection can be accessed using an integer index which is zero-based.

The ArrayList collection accepts an object that refers to Nothing as a valid value, and allows duplicate elements, unlike other types in the Collections namespace.

The following code shows you how to create and manipulate an ArrayList. Copy the code to a button’s Click event handler and run it to observe how the collection functions.

```vbnet
'Declare and create a new ArrayList
Dim myAL As New ArrayList() 'no size required
myAL.Add("Visual")
myAL.Add("Basic")
myAL.Add(".NET")

'Displays the properties and values of the ArrayList
Messagebox.show("myAL has ", myAL.Count.ToString & " elements")
Messagebox.show ("The capacity is ", myAL.Capacity.ToString)

'Remove the extra empty storage
myAL.TrimToSize

Messagebox.show("New Capacity is ", myAL.Capacity.ToString)
Messagebox.show(" Values:")
```

98
Dim obj As Object
    For Each obj in myAL
        MessageBox.Show(obj.ToString)
    Next obj
myAL.Clear() 'removes all members inside the collection

10.2 SortedLists

A **SortedList** works like an **ArrayList** but the advantage of a **SortedList** is that you can use a key for each item added, then search by that key. You may access the elements through their index or associated key. A **SortedList** has the same behavior as an **ArrayList** when it comes to size. As elements are added to a **SortedList**, the capacity is automatically doubled. The capacity can be decreased by calling **TrimToSize** or by setting the **Capacity** property explicitly. The **Count** property shows how many objects are currently in the collection.

**Note** that the elements of a **SortedList** object are sorted by the key, which is why a **SortedList** does not allow duplicate keys, while an **ArrayList** allows duplicate objects because the objects are identified by their index.

The index sequence is based on the sort sequence. When an element is added, it is inserted into **SortedList** in the correct sort order based on its key, and the indexing adjusts accordingly. When an element is removed, the indexing also adjusts accordingly. Therefore, the index of a specific key/value pair might change as elements are added or removed from a **SortedList** object.

A **SortedList** internally maintains two arrays to store the elements of the list: One array for the keys and another array for the associated values. Each element is a key-and-value pair that can be accessed as a **DictionaryEntry** object. This means that the elements of a **SortedList** are of the **DictionaryEntry** type which inherits from **System.Collections**.

**Note** that the key cannot be **Nothing** but a value can be. The methods and properties of a **SortedList** otherwise are similar to an **ArrayList** except for how a **For-Each** statement can work on its members. As you know, a **For-Each** loop requires the type of each element in the collection to be defined. Since each element of the **SortedList** is a key/value pair stored as the **DictionaryEntry** type, make sure you use that type in any **For** loop code as shown below.

' Declare and create a new SortedList
Dim mySL As New SortedList()
mySL.Add("Third: ", ".NET") 'Third is the key, .NET is the value
mySL.Add("Second: ", "Basic")
mySL.Add("First: ", "Visual")
' Displays the properties and values of the SortedList.
    MessageBox.Show("mySL has ", & mySL.Count.ToString & " elements")
    MessageBox.Show("Capacity is ", & mySL.Capacity.ToString)
    MessageBox.Show("Keys and Values ", & KeysNValues(mySL))

mySL.Clear() 'remove all elements from the collection

Function KeysNValues(myList As SortedList) As String
    Dim Result As String = ""
    Result = ".KEY-" & ControlChars.Tab & ".VALUE-" & 
    ControlChars.CR
End Function

99
For Each DE As DictionaryEntry in myList
    Result =& DE.Key.ToString & ControlChars.Tab & _
    myList.Item.ToString & ControlChars.CR
Next
Return Result
End Function

The above code produces output similar to the one shown below. Note that the keys are sorted alphabetically, which is why First shows as the first key value, since F is before S, and S is before T. Try the code above with different keys such as numbers, and negative integers.

    mySL has 3 elements
    Capacity: 16
    Keys and Values:
        -KEY-  -VALUE-
        First: Visual
        Second: Basic
        Third: .NET

10.3 Stacks

A **Stack** represents a simple last-in-first-out collection of ordered objects. You add items to a stack using the **Push** method, and then take them out using the **Pop** method, while the **Peek** method returns the top object without removing it. The **Count** property shows the number of items in the stack. What makes a stack different is that you cannot refer to a member by its index, therefore if you are at the first item, and need to point to the last item then use the **Peek** method for a **Count** number of times. The stack class implements the **IEnumerable**, so if you need to pass a stack object, its placeholder should be declared as **IEnumerable**. Members inside the stack are of the **Object** data type.

The following code shows how to use a stack. It uses a Console application type (as opposed to Windows Forms). Try it for the feel of Console Applications. Note the **Write** and the **WriteLine** methods output content to the screen. Start by creating a new project of the Console Application type. Then you will get a module code view. Type the following code:

```csharp
Imports System
Imports System.Collections

Module Module1

    Sub Main() 'when you click the Start button this sub will be called

        Dim myStack As New Stack()
        myStack.Push("Visual")
        myStack.Push("Basic")
        myStack.Push(".NET")

        Console.WriteLine("myStack")
        Console.WriteLine(ControlChars.Tab & "Count: {0}", myStack.Count)

```
'{} inside the quotes is a place holder to be replaced by an output list
'the output list is the second argument in Writeline
'{} means the first item in the output list provided after the comma

Console.Write(ControlChars.Tab & "Values:" )
PrintValues(myStack)

Console.ReadLine() 'Keep output console open till user hits Enter key
myStack.Clear() 'removes all elements from the stack
End Sub

'The Stack class implements the IEnumerable interface.
' This is why its type is IEnumerable
Sub PrintValues(myCollection As IEnumerable)
For Each myElement As Object In myCollection
    Console.Write(ControlChars.Tab & "{0}", myElement.ToString) 
Loop
Console.WriteLine()
End Sub
End Module

This code above produces the following output.

    myStack
    Count: 3
    Values: .NET Basic Visual

10.4 Queues

A Queue class represents a first-in, first-out collection of objects. Queues are useful for storing messages in the order they were received for sequential processing. Objects stored in a Queue are inserted at one end and removed from the other. If the number of elements added to the Queue reaches the current capacity the capacity is automatically increased to accommodate more elements. Although there is no capacity property to use, the capacity can be decreased by calling TrimToSize. A Queue accepts an object referring to Nothing as a valid value and allows duplicate elements.

To add elements to a queue, the Enqueue method is used, while the Dequeue method is used to remove the current element. The queue may be emptied using the Clear method. The following code sample shows how to manipulate and use queues.

You may add the following code to the code of the Stacks of the previous section.

Sub QuePractice()
    Dim myQ As New Queue()
    myQ.Enqueue("Hello") 'Enqueue adds the element to myQ
    myQ.Enqueue("World")
myQ.Enqueue("!")

' Display the properties and values of the Queue Console.WriteLine("myQ")
Console.WriteLine(ControlChars.Tab & "Count: {0}", myQ.Count)
Console.WriteLine(ControlChars.Tab & "Values:")

PrintValues(myQ) 'calls the same PrintValues of the Stack code above

myQ.Clear() 'removes all elements from the queue

End Sub

Exercises

10.1 Using the same class code for Car and GreenCar, manage the collection using a Stack, and a Queue. Was using a user-defined collection easier to implement? Was using a user-defined collection easier for the end-user?

10.2 A good application for a Queue is handling information in the order it was received. Create a new Window application with a Form that has two textboxes with the Multi-line property set to True, and two buttons as shown below. In real-life applications, the left side would be the end-user screen, and the right-side would an administrator's screen. The end user faces a technical problem and is creating an error report by typing into the right-side textbox, then clicking the Send Message button. The Administrator see's the first message that was sent, and can delete that message once it has been processed. To gage their time, we display how many messages there are as in the form snapshot shown below. Implement such application by adding:

1. One textbox where the client types a message and then clicks the Send Message button.
2. The Send Message adds the text from above to the queue after checking that it's not empty. This updates the label showing how many messages are received so far.
3. The admin side sees a textbox that is empty until the Process Message is clicked which shows the top message received. Once the Admin processes the message they can click the Purge Message which deletes the message from the list and clears the textbox then updates the label with the current number of messages.
Chapter 10

Stacks, Queues and Lists

![Diagram showing a form with two text areas and two buttons: Add Message and Purge Message. The text areas display messages: The machine boots but the screen is blank., The Hard Drive is not booting. There is no sound. My ext is 555. There are 2 messages left.](image)
Chapter 11 Multi-Form Applications

In this chapter, we dive deeper into creating and using multiple forms in a Windows Forms application. Knowing how to create and use forms should follow smoothly after we have learned what a class and an object are.

Now it is time for to learn about Forms as classes and objects. The only difference between employees, cars and forms is that the IDE has a built in feature that makes the form and its controls appear during Design time. These controls and their containing form are nothing but code that gets created once you drag a control to a form. The IDE draws — through a graphics library— the controls on the screen while you work in Design time. Hence, even during design time, the IDE is running code to create instances of these controls and draws them on the screen.

During Design view, the IDE actually creates an instance of the form and the controls that you see painted on the screen. You can, if you like, create a form with controls on it without the Design view. You do so by adding a new class, and then add controls, but you'll be working with code only.

Going back to the Design view, when you are looking at the form's Code view, and trying to understand the content of the Windows generated code, you notice that at the very top you have:

```
Class Form1 'where Form1 is the name you assigned to the form
Inherits Windows...Form 'the last word is the class name.
```

So you are thinking there is a generic class named Form that the current one inherits from. Then you add controls, and that causes the compiler to generate the appropriate code that ‘creates’ instances of these controls and ‘adds’ them to the form. In order to view the code behind the form’s GUI you need to open the form’s GUI code in a text editor such as Notepad. To do so, open the project folder using MyComputer, then point to the form’s .Designer file, and open it in Notepad.

The big question is: During the runtime, which code creates an instance of Form1? The IDE implicitly does it when you choose to create a Windows application. So far, there is no problem anywhere, but here is a problematic scenario.

Start a new Windows Forms Application, and add one textbox on the form. Then add a module using the Project menu, and clicking Add Module. In the code of that module, write code to check the validity of the contents of the textboxes on Form1. For example, write the following:

```
Form1.TextBox1.Text = “Enter Data”
```

Which is a fully valid statement, or is it? Let's look carefully:

Recall that when you worked with the Employee class that in the client code you needed to create an instance of Employee. Here, Form1 is a class name not an instance or object name, and unless TextBox1 is declared as **Shared**, which it is not, then you cannot refer to Form1, but instead, you need to say the ‘Object’ whose type is Form1; recall that Form1 is a type name. But
where is that object? It's not written in code. To solve the problem, you add the line below at the module level:

```
Dim InstForm1 As New Form1
```

Which is completely valid as the line above creates an instance of Form1. So now you need to change how you refer to the textbox into:

```
InstForm1.Textbox1.Text = "Enter Data"
```

But if you try the code above and place a **BreakPoint** (in code-view move the mouse pointer to the far left side and in the grey area click and that creates a brown circle marking the line; during runtime the compiler stops and allows you to monitor the code and run it line by line using the F8 function key). You will notice after you type data in TextBox1 the compiler breaks, and when pointing the mouse on the above line, it shows the content of the textbox to be empty as shown below.

![Visual Studio Debugging](image)

But you know you did type data there. Where did the entry go? Here is what happened:

1. You have a form, which the compiler runs by creating an instance of it internally then pushing that instance onto the screen.
2. On the compiler's instance you type data into the textbox. However, when the module code runs, the compiler (based on your code) creates another instance of Form1, called InstForm1. But,
3. The form you are looking at during Runtime is the compiler’s instance, not your code’s instance.

4. On the instance that your code created no data is typed until line 5 in the figure above finishes execution.

If you created an instance of a form, how come it did not appear on the screen? The reason is when you create an instance and run the code, it is in RAM but not showing on the screen. To show it you must use the Show method of the form class. But that is not all. The compiler’s instance is the one showing without you asking through code. To fix this, you write a sub procedure named Main, as shown below.

```vbnet
Dim InstForm1 As New Form1
Sub Main()
    Application.Run(InstForm1) 'Shows form1 and makes it the startup object
End Sub
```

In addition, you must tell the compiler not to create its own instance of Form1 but to call Sub Main when the start button is clicked instead. What the compiler does when an application is run is called the Startup Object. To make Sub Main the startup object, right click the project icon in the Solution Explorer and click Properties. The Properties screen appears for your current project as shown below.

![Windows Application Properties](image)

First uncheck the Enable Application Framework checkbox – shown on the bottom – then you will see a list of Sub Main, Form1, and Module1 under the Startup object drop down list as shown above. Choose Sub Main, and hit Ok.
11.1 Windows Multi-Forms Projects

In order to create an application that contains multiple forms, you need to decide whether you want to have a main form where all other forms within the application run inside of – called MDI (Multiple Document Interface) application, or simply an application that displays and manipulates multiple independent forms. The latter is simple to execute, as you simply add all the forms you need for the application using the Project menu then Add Windows Form menu. To display a form you use the `FormObjectName.Show()`, and to hide the form you can use `FormObjectName.Hide()`. The MDI approach however is more common, and provides greater control over the application for both programmer and the end-user.

One of the main characteristics of MDI applications is that they contain a main form that has a File and a Window top-level menu items. All other windows may be manipulated through the Window menu. In such applications, the main form is designated to be the Parent form – obviously only one parent form allowed- and all the other forms are considered to be child forms. The main form may appear with its menu, and each child form may bring up its menu. The next section explains how to create a parent and child forms with menus.

11.2 Creating an MDI Project

The MDI parent form is a form that contains MDI child forms, which are the ‘sub-windows’ wherein the user interacts with in the MDI application. To create an MDI application, follow the steps below.

1. Start a new Windows Forms application. The one form you see there will be the parent form. Set its `IsMDIContainer` property from the properties window to `True`. This is your main parent form. Change the Name property to MDIParent.

2. From the Toolbox, drag a `MenuStrip` control to the form. The control will appear in the tray at the bottom of the form, and a placeholder for the menu bar will appear on the top left side of the form.

3. The placeholder will show the text “Click Here”. Click inside and type &File to create a File menu. The ampersand & acts as a hot key the same way it works for buttons.

4. Add more sub-menus under the File menu and type &Data Entry, &Search and &Close.

5. Each item you write inside of, is a control -menu item- by itself. Make sure you change the name property of every menu item to a meaningful name instead of the generic one.

6. While in Design-view each time you click a menu item, you get placeholders to its right and bottom. These placeholders will disappear once you click away from any menu item.

7. Create another top-level menu item called &Window to the right of the File menu. This menu will keep track of the open MDI child forms. The File and Window menus are the minimum menu items needed for an MDI application. The menus of step 4 are for practice.
8. In the drop-down list at the top of the Properties window, select the **MenuStrip** control and then point next to the **MdiWindowListItem** property and click the dropdown list. You will see a listing of the menus you created. Choose the Window menu. This will enable the Window menu to maintain a list of open MDI child windows with a check mark next to the active child window.

9. Add all the other forms to your project. For each form you need to show as a child form, you must do the steps below.

10. On the main form, make sure that either a menu or a control’s **Click** event handler displays a child form. Assuming you have a form object named DataForm and you need to show that form when the File, Data Entry menu item is clicked. After adding the DataForm to the project, add all the needed controls and code to it.

11. Next you need code that displays the DataForm. This is accomplished by adding a **Click** event handler to the Data Entry menu option. While in Design view of the main form, double click the Data Entry menu item; this will open the Code view with the **Click** event handler of that menu item created. Write code to display the Data Entry form using the following:

   ```vbnet
   'The DataEntryMenu item name is based on the Name property of that control
   Private Sub DataEntryMenu_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles DataEntryMenu.Click
       Dim NewMDIChild As New DataForm()
       'Set the Parent Form of the Child window.
       NewMDIChild.MdiParent = Me 'Me is the Parent Form
       NewMDIChild.Show() 'Display the new form
   End Sub
   
   Note that in the code above the child form is used as a Type to create an instance.
   Note also the **MdiParent** property of the child form instance must be set to the parent form. This property is not available during Design-time and must be assigned in code. Also note that since the parent form is the startup object then we don’t know what is the instance or object name that is used to represent the parent form. Since the code above is written inside the parent form class definition, then we may use the **Me** keyword to refer to the parent form.

   Additionally, based on the .NET version you are using, setting the version setting the MDI Parent property of the child form may be done either by setting the property manually or it may be done implicitly once a child form is created.

12. You may repeat steps 10-11 for each child form in the application.

13. Test the application.

### 11.3 Controlling MDI Behavior

Windows automatically gives child forms a default size if their **FormBorderStyle** property is set to a sizable border. This size is based on the size of the parent's client area. You can override this by setting the **FormBorderStyle** property of the child form to any of the fixed type of borders.
The parent form can be minimized and only one icon will be displayed on the desktop representing the parent form and all of its children. Additionally, if the parent form is unloaded, by closing it, all of the loaded children will also be unloaded. The child form may be closed using its close button, or by clicking a menu item to the main form to close the currently open window. You may close a form using the `FormObjectName.Close()` method which removes the form and its code from RAM. If the form is to be redisplayed without being reset, you may use the `FormObjectName.Hide()` and the form will be removed from the screen but not RAM.

It is recommended that you set the `WindowState` property of the parent form to `Maximized` since it is easier to manipulate child forms when the parent form is maximized.

Child forms cannot be displayed modally, that is the child form cannot enforce the focus. Modal forms are like DialogBoxes, where you must dismiss them through making a choice in the DialogBox before it can be closed. You cannot click outside a modal form to switch to another form in the application.

The focus of our work is how to manipulate multi-form applications and how to work with objects of type `Form`. We will not delve any deeper into how to manipulate child and parent forms, much like Microsoft Word behaves, as that is not the focus of our book. The interested reader is referred to www.msdn.microsoft.com for more details.

### 11.4 Working with Window to Control Child Forms

If you have multiple child forms open, and you want to control them as far as arranging and tiling, you may use the following guidelines.

You have different ways to arrange window: Tile Horizontally, Tile Vertically, Cascade and Arrange Icons. To make use of these options under the Window menu, add these items as sub menu choices under the Window menu, then add the Click event handler for each menu item as the following code sample shows. Since the Window menu item was set to be the `MdiWindowListItem` of the application, then setting the `LayoutMDI` property will allow the selected menu item to control how the child windows are arranged. The possible values of the `LayoutMDI` property are an Enumeration called `MDILayout`. The following table shows the possible values:

<table>
<thead>
<tr>
<th>Menu Item</th>
<th>MDILayout Enumerated Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tile Horizontal</td>
<td>MdiLayout.TileHorizontal</td>
</tr>
<tr>
<td>Tile Vertical</td>
<td>MdiLayout.TileVertical</td>
</tr>
<tr>
<td>Cascade</td>
<td>MdiLayout.Cascade</td>
</tr>
<tr>
<td>Arrange Icons</td>
<td>MdiLayout.ArrangeIcons</td>
</tr>
</tbody>
</table>

Assume you added the menu item Horizontalmnu with the text Horizontal as shown below.
The following code example can be adjusted to set menu items to tile windows as needed. To cause the menu item above to arrange all child open windows to be tiled horizontally, double click the Horizontalmnu menu item to open the Code view and add the following code for the Click event handler.

```vbnet
Private Sub Horizontalmnu_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Horizontalmnu.Click
    'Me below refers to the menu item control
    Me.LayoutMdi(MdiLayout.Cascade)
End Sub
```

You may repeat the steps above and add three menu options under the Window menu to allow for Arrange Icons, Tile Vertically and Cascade. However, there is one problem with this approach if there's no active child form.

If there's no active child form, the `Me.ActiveMdiChild` property where `Me` refers to the parent form, will be set to Nothing. You can use this property in an `If` statement to check for existing child forms before attempting to arrange their windows. The following is an example:

```vbnet
If Me.ActiveMdiChild Is Nothing Then
    'some action other than setting the active child form
End If
```

### 11.5 Child Menus

What if a child form has its own set of menus? How do those menus interact with the menus of the parent form? In previous versions of Visual Basic you didn't have much control over the behavior—the menus of the active child simply replaced the menus of its parent. In Visual Studio .NET, however, you can control how the menus interact, using the `MergeIndex` (called
MergeOrder in earlier versions) and MergeAction (called MergeType in earlier versions) properties of the individual menu items. Be aware that if the child form has no menu created, then the parent menu will appear to be the menu of the child form.

To control how a child menu is shown when the parent form has a menu, you need to set the MergeAction and the MergeIndex properties so you can hide the parent’s menu, or decide on the order of the menu items.

Let's look at the MergeAction property first. Keep in mind that you need to set this property for every menu item of both the parent and child menus.

The possible values of MergeAction are the values of an enumeration named MergeAction. The following table lists these values:

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>Menu added to current menubar based on MergeIndex. If both menus MergeIndex</td>
</tr>
<tr>
<td></td>
<td>match, item is added before existing menu.</td>
</tr>
<tr>
<td>Replace</td>
<td>Menu replaces existing menu item in the same position in current menu bar.</td>
</tr>
<tr>
<td></td>
<td>The position is set by the MergeIndex property.</td>
</tr>
<tr>
<td>Append</td>
<td>Adds the menu item at the end of the current menu. MergeIndex value does</td>
</tr>
<tr>
<td></td>
<td>not matter in this case.</td>
</tr>
<tr>
<td>MatchOnly</td>
<td>Matches index of current menu item with to-be-merged menu. MergeAction of</td>
</tr>
<tr>
<td></td>
<td>other menu decides the action.</td>
</tr>
<tr>
<td>Remove</td>
<td>Menu item is not displayed in the menu bar.</td>
</tr>
</tbody>
</table>

The MergeAction property functionality is mostly based on the value of MergeIndex property. Next we will explain how MergeIndex works and how both properties affect child and parent form menus.

The MergeIndex property controls the relative position of the menu item when its menu structure gets merged with the parent form's menus. The default value for this property is 0, indicating that this menu item will be added at the end of the existing menu items of the parent. The MergeAction property controls how the menu item behaves when it has the same MergeIndex as another menu item being merged. Keep in mind that the MergeIndex is a 0-based index. Caution must be taken in setting MergeIndex to -1 as that adds the child menu below the parent menu.

### 11.6 Working with DialogBoxes

A DialogBox is nothing but a form that has its FormBorderStyle property set to FixedDialog in the properties window, and then when showing the form, the ShowDialog method is used instead of the Show method. Make sure that the DialogBox is not a child form, as child forms may not be modal, as explained earlier.

If you need to create a dialogbox just like Inputbox, for example, that returns a value when closed, or a certain button, OK, is clicked, then you need to set a return value, so that the click event handler for that button (OK), closes the form, and returns a value using the Return statement. When coding such dialogboxes, design your code to work with any activating code to show the dialogbox, so you should not write code in the dailogbox form control handler to refer to any code on the main form. For example, if the main form has a control named Text1, do not write a Click event handler for an OK button on the dialogbox to set the Text property of the mainform's Text1. This is not a good programming technique, since the dialogbox in that case will only work with that form.
Design the code so it is general, and use Return statements to make the dialogbox work like Inputbox.

**Exercises**

11.1 In the application developed for a Car application, create a user-defined form that acts as a dialogbox instead of the Inputbox. This dialogbox will used to get the number of miles driven, and later to get the number of gallons pumped. Use this dialogbox to get Miles or Gallons when the Drive or Pump methods are used.

11.2 Update the Collection application so that there is a parent form, from which a child form is shown to start a collection. When a particular car is to be created, or used, use another form (the one created in 11.1). Add menus as needed for parent and child forms.
Chapter 12 Deployment and Assemblies

Now we move to assemblies and application deployment. Application Deployment enables you to create projects that are installable onto client machines. Assemblies on the other hand have to do with the executable format of a project. In the simplest format, this could be the .exe file that is produced when you compile your solution. If your project is just a class code, created in a Class Library application, the assembly is the .dll file. In other cases it may be more than just that.

12.1 Deploying Applications and Components

Deployment is the process by which you distribute a finished product to be installed on other computers. The outcome of a deployment process is a "deployment application". To start a deployment application you choose from different types of deployments based on the type of your original project. You may use a "Merge Module Project" to distribute dll's (class code), or a "Setup Project" for Windows applications, and there are "Web-Setup Projects" as well. The following topics will help you learn more about deployment.

Visual Studio provides templates for four types of deployment projects: Merge Module Project, Setup Project, Web Setup Project, and Cab Project. In addition, a Setup wizard is provided to help step you through creating deployment projects. You can see the templates and the wizard in the New Project dialog box under Other Project Types then Setup and Deployment Projects, as shown below.
**Note** that in the Standard Edition of Visual Basic .NET, only Setup Projects are available. The following table summarizes the different types of deployment project.

<table>
<thead>
<tr>
<th>Project Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Module Project</td>
<td>Packages classes that might be shared by multiple applications. This allows you to package files or components into a single module to allow sharing. The resulting .msm files can be included in any other deployment project. If you have (a) class(es) file(s) that you want to distribute to other machines, then add them in a Merge Module Deployment Project. You may later, need the resulting msm file to be included in another Windows Application, in which case you add the .msm file to the Setup (windows-based) deployment project.</td>
</tr>
<tr>
<td>Cab Project</td>
<td>Creates a cabinet file for downloading to a Web browser. It creates a .cab file to package ActiveX components that can be downloaded from a Web server to a Web browser.</td>
</tr>
<tr>
<td>Setup Project</td>
<td>Builds an installer for a Windows-based application resulting in an .msi file. The distinction between Setup and Web Setup projects is where the installer will be deployed. For a Setup project, the installer will install files into a Program Files directory on a target computer.</td>
</tr>
<tr>
<td>Web Setup Project</td>
<td>Builds an installer for a Web application that will install files into a Virtual Root directory on a Web server such as IIS.</td>
</tr>
</tbody>
</table>

### 12.2 Setup vs. Merge Module

Merge Module projects allow you to create a setup project to install applications on one computer to be shared by other applications. This is useful for packaging classes for development use. In that case, we normally developed a Class Library application and not a Windows Forms application. On the other hand, Setup projects allow you to install applications on different computers through Windows Installers. This is usually done for a completed application that includes an interface such as Forms on end-user machines. Setup produces .msi files while and merge modules produce .msm files.

**Note** Merge Module Projects are not available in the Standard Edition of Visual Basic .NET.

If you are packaging several classes, it is best to put each into its own merge module. While it is possible to put multiple components into a single merge module, it is best to create a merge module for each component in order to avoid distributing unnecessary files.

To create a new merge module project, start a new project of the **Class Library** type, complete the class definition then follow the steps below.

1. On the **File** menu, point to **Add Project**, and then click **New Project**.
2. In the resulting **Add New Project** dialog box, expand the **Other Project Types** then click **Setup and Deployment Projects**.
3. Choose **Merge Module Project**.

To deploy a Windows application, complete the Windows application needed then add a Setup project as shown below.
1. While you have the completed application open, on the **File** menu, point to **Add Project**, and choose **New Project**.

2. In the **Add New Project** dialog box, select **Setup and Deployment Projects** in the **Project Types** pane then choose **Setup Project** in the **Templates** pane. In the **Name** box, type **MyApplication Installer** as shown below.

3. Click **OK** to close the dialog box. The project is added to Solution Explorer, and the File System Editor opens.

4. Select the **MyApplication Installer** project in Solution Explorer. In the Properties window, select the **ProductName** property and type **MyApplication**.

**Note** that the **ProductName** property determines the name that will be displayed for the application in Folder names and in the **Add/Remove Programs** dialog box. The snapshot below shows the **ProductName** property.
Next you need to add the completed application to the installer.

1. Select the **MyApplication Installer** project in Solution Explorer. In the File System Editor, select the **Application Folder** node. Note that the menus will change based on what item you have currently selected.

2. On the **Action** menu, choose **Add, Project Output**.

3. In the **Add Project Output Group** dialog box, choose the application name you want to package from the **Project** drop-down list then select the **Primary Output** group from the list then click **OK**.

4. On the **Build** menu, choose **Build MyApplication Installer**.

It is a good idea to test the deployment application that resulted. To deploy the installer, select the MyApplication Installer project in Solution Explorer and on the **Project** menu, choose **Install**. This will run the installer and installs MyApplication on your computer, assuming you have permissions to install application on the machine.

### 12.3 Assemblies

To better understand this topic, consider the classes that you have developed and compare them to the built-in compiled classes of VB .NET. If you want to use a compiled class that you have created before, you have to copy the compiled files to your application folder through a simple copy process or by running a merge module installer, and then add a reference to it. If you want
to use a compiled class created by Microsoft, on the other hand, you just add a reference to it – there is no need to copy it into your application.

Another situation that requires your better understanding of assemblies is when you think about built-in classes vs. user-defined classes. Think of a situation where you have 3 applications (exe files) running. If each application needs a compiled class that you created, then each application must have a copy of that class file running in its (the application's) folder. But it is not the case if you have 3 applications using one of Microsoft's built-in classes. There is only one compiled class running, and the applications share it. This "sharing" is done through a process called strong naming and signing the assembly (the application files) into the Global Assembly Cache – GAC for short. Learning about assemblies is not only important for you to 'get the big picture' but is the foundation for programming security in VB applications.

Note that in .NET the word assembly (depending on the context and the kind of application you are creating) may mean a specific file named Assembly.vb, or it could simply mean the .exe file of an application. Normally, when you compile and execute code, you get an .exe file that is referred to as the assembly. In large applications (where you might have Type definitions and other code), the compiler generates an assembly file that is more than a single executable file.

So what does an assembly represent? If you think of an application as a collection of form files, code module files and Class files, then to "start" or run the entire application, an Assembly is the "director" of the application that checks all the files, data types, and locations of these files. Assemblies are the building blocks of .NET Framework applications; they form the fundamental unit of deployment. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. It also provides the runtime with the information it needs to be aware of type implementations. To the runtime, a type does not exist outside the context of an assembly.

In general, an assembly may consist of four elements:

- The assembly manifest, which is meta data about how the assembly content is related. The assembly manifest can be stored in either a PE file (an .exe or .dll) with Microsoft intermediate language (MSIL) code or in a standalone PE file that contains only assembly manifest information.
- Type information or metadata.
- Microsoft intermediate language (MSIL) code that implements the types.
- A set of resources.

12.3.1 Creating Assemblies

You can create single-file or multi-file assemblies using an IDE, such as Visual Studio .NET, or the compilers and tools provided by the .NET Framework SDK. The simplest assembly is a single file that has a simple name -as opposed to a strong name covered later in this chapter- and is loaded into a single application domain -as opposed to being shared by other applications. This assembly cannot be referenced by other assemblies outside the application directory. To uninstall the application made up of this assembly, you simply delete the directory where it resides. For many developers, an assembly with these features is all that is needed to deploy an application.

Alternatively, the elements of an assembly can be contained in several files. These files can be modules of compiled code, resources (such as .bmp or .jpg files), or other files required by the application. You should create a multi-file assembly when you want to combine modules written
in different languages and to optimize downloading an application by putting seldom used types in a module that is downloaded only when needed.

You can also create an assembly that can be shared by multiple applications. A shared assembly must have a strong name and can be deployed in the global assembly cache, GAC. The next section demonstrates how to create a strong name and how to deploy an assembly in the GAC, but first we need to explain what is GAC.

**12.4 The Global Assembly Cache**

The .NET Framework global assembly cache is a code cache that is automatically installed with any installation of the .NET Framework common language runtime. Any application that is installed on a computer that has the .NET Framework installed can access the GAC. The GAC stores assemblies that are specifically designated to be shared by several applications on the computer. These to-be-shared files are typically stored in the C:\WINNT\Assembly folder.

It is therefore recommended that you only install an assembly in the GAC when you must share the assembly. Otherwise just install the needed assembly as part of a merge module installer.

**12.5 Strong Names**

An assembly must have a strong name to be installed in the global assembly cache. A strong name is a unique identity that identifies the assembly as safe. By using a strong name, you prevent components that have the same name from conflicting with each other or from being used incorrectly by a calling application. A strong name consists of the following information:

- The name of the assembly.
- The version number of the assembly.
- The culture information about the assembly, if it is provided.
- A public key and private key pair that form the unique identity.

This information is stored in a key file. The key file is either a Personal Information Exchange (.pfx) file or a certificate from the current user's Microsoft Windows certificate store.

You can sign an assembly by using the options on the Signing tab of the Project Designer. The key file must be stored in the project folder on the local computer. Visual Studio .NET supports only the following file formats:

- Personal Information Exchange (.pfx) files.
- Strong name key (.snk) files.

Before you attempt to install an assembly in GAC, ensure that you satisfy the following:

- You must have Administrator rights to the computer where the shared assembly is installed.
- You must install the .NET Framework SDK (search www.microsoft.com for the SDK's download site –this is a free download). The SDK is a Command prompt software that allows you to compile source code and execute several other systems commands.

**12.6 Installing an Assembly in the GAC**

To create an assembly that can be shared by multiple applications, the shared assembly must have a strong name. Then, the shared assembly must be deployed in the global assembly cache.
To create an assembly that has a strong name and to install the resulting compiled .dll file in the global assembly cache, follow these steps:

1. To generate a strong name, and then associate the strong name key file together with the assembly follow these steps:
   a. On the Project menu, click GACDemo Properties.
   b. On the Signing tab, click to select the Sign the assembly check box.
   c. Under Choose a strong name key file, click <New>.
   d. In the Create Strong Name Key dialog box, click to select the Protect my key file with a password check box.
   e. In the Key file name box, type GACDemo.
   f. In the Enter password box, type the password that you want to use.
   g. In the Confirm password box, type the same password, and then click OK.
      Note We recommend that you always use a password when you create a key file. A new key file that is protected by a password is always created in the .pfx file format.
   h. To compile the project, press CTRL+SHIFT+B.
      Note No additional code is required to install a .dll file in the global assembly cache.

2. Install the .dll file that you created in step 2 in the global assembly cache by using the Global Assembly Cache tool. To do this, follow these steps:
   a. Click Start, click Run, type cmd, and then click OK.
   b. To change the current working directory to the directory where the .NET Framework SDK is installed, use one of the following commands:
      i. If the .NET Framework version 2.0 SDK is installed, type the following command, and then press ENTER:
         cd "%ProgramFiles%\Microsoft Visual Studio 8\SDK\v2.0\Bin"
      ii. If the .NET Framework version 1.1 SDK is installed, type the following command, and then press ENTER:
         cd "%ProgramFiles%\Microsoft.NET\SDK\v1.1\Bin"
   c. At a command prompt, type the following command, and then press ENTER:
      gacutil -l GACDemo
To verify that the assembly is installed in the global assembly cache you can use the Global Assembly Cache tool to verify that the assembly is installed in the global assembly cache by following these steps:

1. Click Start, click Run, type cmd, and then click OK.
2. To change the current working directory to the directory where the .NET Framework SDK is installed, use one of the following commands:
   a. If the .NET Framework 2.0 SDK is installed, type the following command, and then press ENTER:
      cd "%ProgramFiles%\Microsoft Visual Studio 8\SDK\v2.0\Bin"
   b. If the .NET Framework 1.1 SDK is installed, type the following command, and then press ENTER:
      cd "%ProgramFiles%\Microsoft.NET\SDK\v1.1\Bin"
3. To display the installation information about the GACDemo assembly, use the Global Assembly Cache tool. To do this, type the following command at a command prompt, and then press ENTER:
   gacutil -l GACDemo

   Note The installation information about the GACDemo assembly is displayed.
12.7 The Common Type System and Closing Remarks

Before we end our journey into VB’s OOP world, it is important to understand how VB classifies and uses the different built-in fundamental types vs. the object-based types. A component of the .NET language uses what is called the Common Type System to classify and manage the different types.

The common type system supports two general categories of types, each of which is further divided into subcategories:

- Value types that directly contain their data.
- Reference types, which store a memory address where the value is stored.

Variables that are value types each have their own copy of the data, and therefore operations on one variable do not affect other variables. Variables that are reference types can refer to the same object; therefore, operations on one variable can affect the same object referred to by another variable.

The following example shows the difference between reference types and value types.

```vbnet
Imports System
Class Class1
    Public Value As Integer = 0
End Class 'Class1
Class Test
    Shared Sub Main()
        Dim val1 As Integer = 0
        Dim val2 As Integer = val1
        val2 = 123
        Dim ref1 As New Class1()
        Dim ref2 As Class1 = ref1
        ref2.Value = 123
        Console.WriteLine("Values: {0}, {1}", val1, val2)
        Console.WriteLine("Refs: {0}, {1}", ref1.Value, ref2.Value)
    End Sub 'Main
End Class 'Test
```

The output from this program is as follows.

Values: 0, 123
Refs: 123, 123

In the output above, both val1 and val2 are Integer variables of a value type. Even though one variable is assigned to the other, any future reassignment of one of the variables, does not affect the content of the other variable. Hence when val1 in memory contains 0, then next val2 is assigned to val1, causing val2 to contain 0. Because both val1 and val2 are value types, their memory locations contain 0. Later when val2 is assigned to 123, the content of val2 is changed to 123, while val1 is not affected, since it is not connected in any way to val1.

When it comes to ref1 and ref2 that are both object types - references types- when ref2 is assigned to ref1, then the memory content of ref2 contains the memory address of where ref1 is.
Hence if the content of ref1 changes, then ref2 which points to the same content, will also appear different.

The following table shows example of different types, and where they stand.

<table>
<thead>
<tr>
<th>Value Types</th>
<th>Reference Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>All numeric data types, Boolean, Char and Date in addition all Enumerations.</td>
<td>String, Arrays of any type, Class types, whether built-in or user-defined.</td>
</tr>
</tbody>
</table>