PROGRAMMING

Introduction to Object-Oriented Programming

With Visual Basic.NET

Ibrahim Otieno
CHAPTER ONE

Introduction to .NET framework

1.1 What is Visual Basic .NET?

Visual Basic .NET is part of a grand new initiative by Microsoft. It is a complete re-engineering of Visual Basic for the Microsoft .NET framework. With Visual Basic .NET, you are able to quickly build Windows-based applications (the emphasis in this course), web-based applications and, eventually, software for other devices, such as palm computers.

Windows applications built using Visual Basic .NET feature a Graphical User Interface (GUI). Users interact with a set of visual tools (buttons, text boxes, tool bars, and menu items) to make an application do its required tasks. The applications have a familiar appearance to the user. As you develop as a Visual Basic .NET programmer, you will begin to look at Windows applications in a different light. You will recognize and understand how various elements of Word, Excel, Access and other applications work. You will develop a new vocabulary to describe the elements of Windows applications.

Visual Basic .NET Windows applications are event-driven, meaning nothing happens until an application is called upon to respond to some event (button pressing, menu selection,). Visual Basic .NET is governed by an event processor. As mentioned, nothing happens until an event is detected. Once an event is detected, a corresponding event procedure is located and the instructions provided by that procedure are executed. Those instructions are the actual code written by the programmer. In Visual Basic .NET, that code is written using a version of the BASIC programming language. Once an event procedure is completed, program control is then returned to the event processor.

Event?

Event processor

Basic Code

Basic Code

Basic Code

All Windows applications are event-driven. For example, nothing happens in Word until you click on a button, select a menu option, or type some text. Each of these actions is an event.

The event-driven nature of applications developed with Visual Basic .NET makes it very easy to work with. As you develop a Visual Basic .NET application, event procedures can be built and tested individually, saving development time. And, often event procedures are similar in their coding, allowing re-use (and lots of copy and paste).

Some Features of Visual Basic .NET

- All new, easy-to-use, powerful Integrated Development Environment (IDE)
- Full set of controls - you 'draw' the application
- Response to mouse and keyboard actions
- Clipboard and printer access
- Full array of mathematical, string handling, and graphics functions
- Can easily work with arrays of variables and objects
- Sequential file support
- Useful debugger and structured error-handling facilities
- Easy-to-use graphic tools
- Powerful database access tools
- Ability to develop both Windows and internet applications using similar techniques
- New common language runtime module makes distribution of applications a simple task
1.2. Visual Basic .NET versus Visual Basic

Let's get something straight right now – Visual Basic .NET is not a new version of Visual Basic. Visual Basic .NET is an entirely new product. If you are familiar with Visual Basic, Visual Basic .NET will look familiar, but there are many differences. And, for the most part, the differences are vast improvements over Visual Basic. When you realize that the BASIC language has not undergone substantial changes in 20 years, you should agree it was time for a clean-up and improvement.

A few of the features of Visual Basic .NET, compared to Visual Basic:

- New Integrated Development Environment
- Uses Object-Oriented Programming (OOP) methods
- New controls and control properties
- Redesigned code window
- Zero-based arrays (no adjustable first dimension)
- Easier to use common dialog boxes
- Structured error-handling (no more On Error Go To)
- New menu design tools
- New techniques for working with sequential files
- All new graphics methods
- New approaches to printing from an application
- Improved support to incorporating help systems in applications
- New web forms for internet applications
- ADO.NET for database access

To use Visual Basic .NET, you (and your users) must be using Windows 2000, Windows XP, or Windows NT. As of now, Visual Basic .NET applications will not run on Windows 9X or Windows Me machines.


As of this writing, the current version of Visual Basic .NET is referred to as Visual Basic .NET 2003 (part of Visual Studio .NET 2003). This new version is an upgrade to the original Visual Basic .NET, now referred to by some as Visual Basic .NET 2002. You can complete this course with either version. Where there are differences in the versions, those differences will be explained. The code examples provided with the notes were developed using Visual Basic .NET, but can be used with Visual Basic .NET 2003.

Some new features and additions of Visual Basic .NET 2003:

- New development environment called the Microsoft Development Environment.
- Fixes many of the bugs found in Visual Basic .NET 2002 and provides easy upgrade of Visual Basic .NET 2002 programs.
- Includes support to develop applications for more than 200 mobile Web devices.
- New data providers for easy connection to Microsoft SQL Server, Microsoft Access, Jet, DB2, Oracle, and more.
- New .NET Framework.
- Enhanced XML Web services support.
- On-line third party community websites.
- Enhanced Intellisense for code completion.
- Improved debugger.
1.4. A Brief Look at Object-Oriented Programming (OOP)

Since Visual Basic was first introduced in the early 1990’s, a major criticism from many programmers (especially those using C and C++) was that it was not a true object-oriented language. And, with that limitation, many dismissed Visual Basic as a “toy” language. That limitation no longer exists!

Visual Basic .NET is fully object-oriented. For this particular course, we don’t have to worry much about just what that means (many sizeable tomes have been written about OOP). What we need to know is that each application we write will be made up of objects. Just what is an object? It can be many things: a variable, a font, a graphics region, a rectangle, a printed document. The key thing to remember is that these objects represent reusable entities that are used to develop an application. This ‘reusability’ makes our job much easier as a programmer.

In Visual Basic .NET, there are three terms we need to be familiar with in working with object-oriented programming: Namespace, Class and Object. Objects are what are used to build our application. We will learn about many objects throughout this course. Objects are derived from classes. Think of classes as general descriptions of objects, which are then specific implementations of a class. For example, a class could be a general description of a car, where an object from that class would be a specific car, say a red 1965 Ford Mustang convertible (a nice object!). Lastly, a namespace is a grouping of different classes used in the .NET world. One namespace might have graphics classes, while another would have math functions. We will see several namespaces in our work.

When seeking help, you need to know that an object comes from a class which comes from a namespace. We will further delve into the world of OOP in subsequent chapters. Then, you’ll be able to throw around terms like inheritance, polymorphism, overloading, encapsulation, and overriding.

The biggest advantage of the object-oriented nature of Visual Basic .NET is that it is no longer a “toy” language. In fact, Visual Basic .NET uses the same platform for development and deployment (incorporating the new Common Language Runtime (CLR) module as the more esoteric languages (Visual C++ and the new Visual C#). Because of this, there should be no performance differences between applications written in Visual Basic .NET, Visual C++ .NET, or Visual C# .NET!

1.5. Microsoft.NET Framework

The Microsoft .NET Framework is a newly established software development environment which helps developers to develop applications quickly and gives optimum, efficient, scalable, performance oriented applications in different languages like Visual Basic .NET, C#, ASP .NET, and Jscript .NET etc…

The Microsoft .NET Framework is a component of the Microsoft Windows operating system. It provides a large body of pre-coded solutions to common program requirements, and manages the execution of programs written specifically for the framework. The .NET Framework is a key Microsoft offering, and is intended to be used by most new applications created for the Windows platform.

The pre-coded solutions form the framework's class library and cover a large range of programming needs in areas including the user interface, data access, cryptography, numeric algorithms, and network communications. The functions of the class library are used by programmers who combine them with their own code to produce applications.

Programs written for the .NET framework execute in a software environment that manages the program's runtime requirements. This runtime environment, which is also a part of the .NET framework, is known as the Common Language Runtime (CLR). The CLR provides the appearance of an application virtual machine, so that programmers need not consider the capabilities of the specific CPU that will execute the program. The CLR also provides other important services such as security guarantees, memory management, and exception handling.

The class library and the CLR together comprise the .NET framework. The framework is intended to make it easier to develop computer applications and to reduce the vulnerability of applications and computers to security threats.
The .NET Framework was designed with several intentions:

- **Interoperability** - Because so many COM libraries have already been created, the .NET Framework provides methods for allowing interoperability between new code and existing libraries.

- **Common Runtime Engine** - Programming languages on the .NET Framework compile into an intermediate language known as the Common Intermediate Language, or CIL; Microsoft's implementation of CIL is known as Microsoft Intermediate Language, or MSIL. In Microsoft's implementation, this intermediate language is not interpreted, but rather compiled in a manner known as just-in-time compilation (JIT) into native code. The combination of these concepts is called the Common Language Infrastructure (CLI), a specification; Microsoft's implementation of the CLI is known as the Common Language Runtime (CLR).

- **Language Independence** - The .NET Framework introduces a Common Type System, orCTS. The CTS specification defines all possible data types and programming constructs supported by the CLR and how they may or may not interact with each other. Because of this feature, the .NET Framework supports development in multiple programming languages.

- **Base Class Library** - The Base Class Library (BCL), sometimes referred to as the Framework Class Library (FCL), is a library of types available to all languages using the .NET Framework. The BCL provides classes which encapsulate a number of common functions such as file reading and writing, graphic rendering, database interaction, XML document manipulation, and so forth.

- **Simplified Deployment** - Installation and deployment of Windows applications has been the bane of many developers' existence. Registry settings, file distribution and DLL hell have been nearly completely eliminated by new deployment mechanisms in the .NET Framework.

- **Security** - .NET allows for code to be run with different trust levels without the use of a separate sandbox.

The design of the .NET framework is such that it supports platform independence. That is, a program written to use the framework should run without change on any platform for which the framework is implemented. At present, Microsoft has implemented the full framework only on the Windows operating system.
1.7 Microsoft.NET Framework Services

NET Framework provides the following services:

- Tools for developing software applications,
- run-time environments for software application to execute,
- server infrastructure,
- value added intelligent software which helps developers to do less coding and work efficiently,

The .Net Framework will enable developers to develop applications for various devices and platforms like windows application, web applications windows services and web services.

1.8. Objectives of Microsoft.NET Framework

The .NET framework is designed to fulfill the following objectives:

- A consistent object-oriented programming environment, where object code can be stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- A code-execution environment that minimizes software deployment and versioning conflicts.
- A code-execution environment that guarantees safe execution of code, including code created by an unknown or semi-trusted third party.
- A code-execution environment that eliminates the performance problems of scripted or interpreted environments.
- Developers can experience consistency across widely varying types of applications, such as Windows-based applications and Web-based applications.
- Build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code.

1.9. Structure of a Visual Basic .NET Windows Application

We want to get started building our first Visual Basic .NET Windows application. But, first we need to define some of the terminology we will be using. In Visual Basic .NET, a Windows application is defined as a solution. A solution is made up of one or more projects. Projects are groups of forms and code that make up some application. In most of our work in this course, our applications (solutions) will be made up of a single project. Because of this, we will usually use the terms application, solution and project synonymously.

Application (Project) is made up of:

- **Forms** - Windows that you create for user interface
- **Controls** - Graphical features drawn on forms to allow user interaction (text boxes, labels, scroll bars, buttons, etc.) (Forms and Controls are objects.)
- **Properties** - Every characteristic of a form or control is specified by a property. Example properties include names, captions, size, color, position, and contents. Visual Basic .NET applies default properties. You can change properties when designing the application or even when an application is executing.
- **Methods** - Built-in procedures that can be invoked to impart some action to a particular object.
- **Event Procedures** - Code related to some object or control. This is the code that is executed when a certain event occurs. In our applications, this code will be written in the BASIC language.
- **General Procedures** - Code not related to objects. This code must be invoked or called in the application.
- **Modules** - Collection of general procedures, variable declarations, and constant definitions used by an application.
Visual Basic .NET uses a very specific directory structure for saving all of the components for a particular application. When you start a new project (solution), you will be asked for a Name and Location (directory). A folder named Name will be established in the selected Location. That folder will be used to store all solution files, project files, form and module files (vb extension) and other files needed by the project. Two subfolders will be established within the Name folder: Bin and Obj. The Obj folder contains files used for debugging your application as it is being developed. The Bin folder contains your compiled application (the actual executable code or exe file). Later, you will see that this folder is considered the ‘application path’ when ancillary data, graphics and sound files are needed by an application.

In a project folder, you will see these files (and possibly more):

- **AssemblyInfo.vb**: Information on how things fit together
- **SolutionName.sln**: Solution file for solution named SolutionName
- **ProjectName.vbproj**: Project file – one for each project in solution
- **ProjectName.vbproj.user**: Another Project file – one for each project in solution
- **FormName.resx**: Form resources file – one for each form
- **FormName.vb**: Form code file – one for each form

The **Solution Explorer Window** displays a list of all forms, modules and other files making up your application. To view a form in this window, simply double-click the file name. Or, highlight the file and press <Shift>-<F7>. Or, you can obtain a view of the Form or Code windows (window containing the actual basic coding) from the Project window, using the toolbar near the top of the window. As we mentioned, there are many ways to do things using the Visual Basic .NET IDE.

Two windows useful when running an application are the **Task List Window** and **Output Window**. The task list window highlights any errors encountered while trying to run an application. The window will direct you to the line(s) of code with identified errors. The output window outlines the steps followed as an application is compiled.
CHAPTER TWO

Basic Programming in Visual Basic.NET

2.1 Variables

We’re now ready to write code for our application. As just seen, as controls are added to the form, Visual Basic .NET builds a framework of event procedures. We simply add code to the event procedures we want our application to respond to. But before we do this, we need to discuss variables.

Variables are used by Visual Basic .NET to hold information needed by an application. Variables must be properly named. Rules used in naming variables:

- No more than 40 characters
- They may include letters, numbers, and underscore (_)
- The first character must be a letter
- You cannot use a reserved word (keywords used by Visual Basic .NET)

Use meaningful variable names that help you (or other programmers) understand the purpose of the information stored by the variable.

Examples of acceptable variable names:

StartingTime  Interest_Value  Letter05
JohnsAge      Number_of_Days  TimeOfDay

2.2 Visual Basic .NET Data Types

Each variable is used to store information of a particular type. Visual Basic .NET has a wide range of data types. You must always know the type of information stored in a particular variable.

Boolean variables can have one of two different values: True or False (reserved words in Visual Basic .NET). Boolean variables are helpful in making decisions.

If a variable stores a whole number (no decimal), there are three data types available: Short, Integer or Long. Which type you select depends on the range of the value stored by the variable:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short</td>
<td>-32,678 to 32,767</td>
</tr>
<tr>
<td>Integer</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>Long</td>
<td>-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807</td>
</tr>
</tbody>
</table>

We will almost always use the Integer type in our work.

If a variable stores a decimal number, there are two data types: Single or Double. The Double uses twice as much storage as Single, providing more precision and a wider range. Examples:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>3.14</td>
</tr>
<tr>
<td>Double</td>
<td>3.14159265359</td>
</tr>
</tbody>
</table>
Visual Basic .NET is a popular language for performing string manipulations. These manipulations are performed on variables of type **String**. A string variable is just that - a string (list) of various characters. In Visual Basic .NET, string variable values are enclosed in quotes. Examples of string variables:

```
“Visual Basic .NET”
“012345”
“Title Author”
```

Single character string variables have a special type, type **Char**, for character type. Examples of character variables:

```
“a”
“1”
“¥”
“#”
```

Visual Basic .NET also has great facilities for handling dates and times. The **Date** variable type stores both date and time. Using different formatting techniques (we will learn these) allow us to display dates and times in any format desired.

A last data type is type **Object**. That is, we can actually define a variable to represent any Visual Basic .NET object, like a button or form. We will see the utility of the Object type as we progress in the course.

### 2.3 Variable Declaration

Once we have decided on a variable name and the type of variable, we must tell our Visual Basic .NET application what that name and type are. We say, we must **explicitly declare** the variable.

It is possible to use Visual Basic .NET without declaring variables, but this is a dangerous practice. There are many advantages to **explicitly** typing variables. Primarily, we insure all computations are properly done, mistyped variable names are easily spotted, and Visual Basic .NET will take care of insuring consistency in variable names. Because of these advantages, and because it is good programming practice, we will **always** explicitly type variables. To insure variable typing is checked, we will **always** add this single line to the top of every application’s code:

```
Option Explicit On
```

To **explicitly** type a variable, you must first determine its **scope**. Scope identifies how widely disseminated we want the variable value to be. We will use four levels of scope:

- Procedure level
- Procedure level, static

---

Visual Basic .Net
• Form and module level
• Global level

2.3.1 Procedure level
The value of a procedure level variable is only available within a procedure. Such variables are declared within a procedure using the Dim statement:

Dim MyInt as Integer

Dim MyDouble as Double

Dim MyString As String, YourString as String

Procedure level variables declared in this manner do not retain their value once a procedure terminates. To make a procedure level variable retain its value upon exiting the procedure; replace the Dim keyword with Static:

Static MyInt as Integer
Static MyDouble as Double

2.3.2 Form level

Form (module) level variables retain their value and are available to all procedures within that form (module). Form (module) level variables are declared in the code window right after the code generated automatically be Visual Basic .NET (the Windows Form Designer generated code).

The Dim keyword is used to declare form (module) level variables:

Dim MyInt as Integer
Dim MyDate as Date
2.3.3 Global level

Global level variables retain their value and are available to all forms, modules and procedures within an application. Such variables can only be declared in a code module – a separate project component (discussed in Chapter 5). Global level variables are declared in a module's code window prior to any procedure. (It is advisable to keep all global variables in one module.) Use the Public keyword:

Public MyInt as Integer

Public MyDate as Date

What happens if you declare a variable with the same name in two or more places? More local variables shadow (are accessed in preference to) less local variables. For example, if a variable MyInt is defined as Public in a module and declared local in a routine MyRoutine, while in MyRoutine, the local value of MyInt is accessed. Outside MyRoutine, the global value of MyInt is accessed.

Example of Variable Scope:

Module1

Public X as Integer

Form1

Dim Y as Integer

Sub Routine1()
    Dim A as Double
    .
    End Sub

Sub Routine2()
    Static B as Double
    .
    End Sub

Form2

Dim Z as Single

Sub Routine3()
    Dim C as String
    .
    End Sub

Procedure Routine1 has access to X, Y, and A (loses value upon termination)
Procedure Routine2 has access to X, Y, and B (retains value)
Procedure Routine3 has access to X, Z, and C (loses value)

2.4 Arrays

Visual Basic .NET has powerful facilities for handling arrays, which provide a way to store a large number of variables under the same name. Each variable, called an element, in an array must have the same data type, and they are distinguished from each other by an array index. In this class, we work with one-dimensional arrays, although multi-dimensional arrays are possible.
Arrays are declared in a manner identical to that used for regular variables. For example, to declare an integer array named 'Item', with dimension 9, at the procedure level, we use:

```visualbasic
Dim Item(9) as Integer
```

If we want the array variables to retain their value upon leaving a procedure, we use the keyword `Static`:

```visualbasic
Static Item(9) as Integer
```

At the form or module level, in the proper region of the Code window, use:

```visualbasic
Dim Item(9) as Integer
```

And, at the global level, in a code module, use:

```visualbasic
Public Item(9) as Integer
```

The index on an array variable begins at 0 and ends at the dimensioned value. Hence, the `Item` array in the above examples has ten elements, ranging from Item(0) to Item(9). You use array variables just like any other variable - just remember to include its name and its index. Many times, the 0 index is ignored and we just start with item 1. But sometimes (especially when working with Visual Basic.NET controls) the 0th element cannot be ignored. You will see examples of both 0-based and 1-based arrays in the course examples.

It is also possible to have arrays of controls. For example, to have 20 button types available use:

```visualbasic
Dim MyButtons(19) as Button
```

### 2.5 Constants

You can also define constants for use in Visual Basic .NET. The format for defining a constant named `NumberOfUses` with a value 200 is:

```visualbasic
Const NumberOfUses As Integer = 200
```

The scope of user-defined constants is established the same way a variables’ scope is. That is, if defined within a procedure, they are local to the procedure. If defined in the top region of a form’s code window, they are global to the form. To make constants global to an application, use the format within a code module:

```visualbasic
Public Const NumberOfUses As Integer = 200
```

If you attempt to change the value of a defined constant, your program will stop with an error message.

### 2.6 Variable Initialization

By default, any declared numeric variables are initialized at zero. String variables are initialized at an empty string. If desired, Visual Basic .NET lets you initialize variables at the same time you declare them. Just insure that the type of initial value matches the variable type (i.e. don’t assign a string value to an integer variable).
Examples of variable initialization:

```
Dim MyInt as Integer = 23
Dim MyString as String = “Visual Basic .NET”
Static MyDouble as Double = 7.28474646464
Public MyDate as Date = #1/2/2001#
```

You can even initialize arrays with this technique. You must, however, delete the explicit dimension (number of elements) and let Visual Basic .NET figure it out by counting the number of elements used to initialize the array. An example is:

```
Dim Item() as Integer = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Visual Basic .NET will know this array has 10 elements (a dimension of 9).

### 2.7 Intellisense Feature

Yes, we're finally ready to start writing some code in the code window. You will see that typing code is just like using any word processor. The usual navigation and editing features are all there.

One feature that you will become comfortable with and amazed with is called **Intellisense**. As you type code, the Intellisense feature will, at times, provide assistance in completing lines of code. For example, once you type a control name and a dot (.), a drop-down list of possible properties and methods will appear. When we use functions and procedures, suggested values for arguments will be provided. Syntax errors will be identified. And, potential errors with running an application will be pointed out.

Intellisense is a very useful part of Visual Basic .NET. You should become acquainted with its use and how to select suggested values. We tell you about now so you won’t be surprised when little boxes start popping up as you type code.

### 2.8 Exercises

**Exercise 2-1 – Practice**

Write a program that:

- Returns the number of times a character appears in a string
- Returns the a string typed by the user in reverse
- Computes the factorial of a given number input by the user
- Computes the a number raised to a given power
**Exercise 2-2 - Stopwatch Application**

Create an application with three buttons and six labels that simulates the function of a stop watch. The interface should be as shown below:

![Stopwatch Application interface](image)

**Exercise 2-3 - Beep problem.**

Build an application with a single button. When the button is clicked, make the computer beep (use the `Beep` function).

**Exercise 2-4 - Text Problem**

Build an application with a single button. When the button is clicked, change the button’s `Text` property. This allows a button to be used for multiple purposes. If you want to change the button caption back when you click again, you’ll need an `If` statement. We’ll discuss this statement in the next class, but, if you’re adventurous, look in on-line help to try it.

**Exercise 2-5 Enable problem**

Build an application with two buttons. When you click one button, make it disabled (`Enabled = False`) and make the other button enabled (`Enabled = True`).

**Exercise 2-6 Date problem**

Build an application with a button. When the button is clicked, have the computer display the current date in a label control.
CHAPTER THREE

The Visual Basic .NET Language

3.1 A Brief History of BASIC

The BASIC language was developed in the early 1960's at Dartmouth College as a device for teaching programming to “ordinary” people. There is a reason it’s called BASIC:

- B (Beginner's)
- A (All-Purpose)
- S (Symbolic)
- I (Instruction)
- C (Code)

When timesharing systems were introduced in the 1960's, BASIC was the language of choice. Many of the first computer simulation games (Star Trek, for example) were written in timeshare BASIC. In the mid-1970's, two college students decided that the new Altair microcomputer needed a BASIC language interpreter. They sold their product on cassette tape for a cost of $350. You may have heard of these entrepreneurs: Bill Gates and Paul Allen!

Every BASIC written since then has been based on that early version. Examples include: GW-Basic, QBasic, and QuickBasic. Visual Basic (allowing development of Windows applications) was first introduced in 1991. The latest (and last) version of Visual Basic is Version 6.0, which was released in 1997.

Visual Basic .NET still uses the BASIC language to write code. Visual Basic .NET provides a long overdue enhancement to the language with many new features. In addition, many archaic features than have been around since Bill and Paul's earliest efforts have finally been retired. This chapter provides an overview of the BASIC language used in the Visual Basic .NET environment.

3.2 Visual Basic .NET Statements and Expressions

The simplest (and most common) statement in Visual Basic .NET is the assignment statement. It consists of a variable name, followed by the assignment operator (=), followed by some sort of expression. The expression on the right hand side is evaluated, then the variable (or property) on the left hand side of the assignment operator is replaced by that value of the expression.

Examples:

```
lblExplorer.Text = "Captain Spaulding"
BitCount = ByteCount * 8
Energy = Mass * LIGHTSPEED ^ 2
NetWorth = Assets - Liabilities
```

Statements normally take up a single line with no terminator. Statements can be stacked by using a colon (:) to separate them. Example:

```
StartTime = Now: EndTime = StartTime + 10
```

The above code is the same as if the second statement followed the first statement. Be careful stacking statements, especially with If/End If structures (we'll learn about these soon). You may not get the response you desire. The only place we tend to use stacking is for quick initialization of like variables.
If a statement is very long, it may be continued to the next line using the **continuation** character, an underscore (_). Example:

```
Months = Math.Log(Final * IntRate / Deposit + 1) _
/ Math.Log(1 + IntRate)
```

We don’t use continuation statements very much in these notes or in our examples. Be aware that long lines of code in the notes many times wrap around to the next line (due to page margins).

### 3.3 Comments

Comment statements begin with the keyword **Rem** or a single quote (’). For example:

```
Rem This is a remark
' This is also a remark
x = 2 * y ' another way to write a remark or comment
```

You, as a programmer, should decide how much to comment your code. Consider such factors as reuse, your audience, and the legacy of your code. In our notes and examples, we try to insert comment statements when necessary to explain some detail.

### 3.4 Strict Type Checking

In each assignment statement, it is important that the type of data on both sides of the operator (=) is the same. That is, if the variable on the left side of the operator is an **Integer**, the result of the expression on the right side should be **Integer**.

Visual Basic .NET (by default) will try to do any conversions for you. Sometimes, however, it ‘guesses’ incorrectly and may provide incorrect or undesired results. To insure consistency of data types in assignment statements, Visual Basic .NET offers, as an option, **strict type checking**. What this means is that your program will not run unless each side of an assignment operator has the same type of data. With strict type checking, the Intellisense feature of the Code window will identify where there are type inconsistencies and provide suggestions on how to correct the situation.

Strict type checking will force you to write good code and eliminate many potential errors. Because of this, all examples for the remainder of this course will use strict type checking. To turn on strict type checking, we place a single line of code at the top of each application:

```
Option Strict On
```

This line will go right below the **Option Explicit On** line which forces us to declare each variable we use.

Remember, **every** application should have the above two lines at the top of the code window. So, as a rule, every new application’s code window should immediately be changed to appear like this:
This will enforce both explicit variable declaration and type checking across the assignment operator, both very good programming practices.

Strict type checking also means we need ways to convert one data type to another. Visual Basic .NET offers a wealth of functions that perform these conversions. Some of these functions are:

- **CBool(Expression)**: Converts a numerical Expression to a Boolean data type (if Expression is nonzero, the result is True, otherwise the result is False)
- **CChar(Expression)**: Converts any valid single character string to a Char data type
- **CDate(Expression)**: Converts any valid date or time Expression to a Date data type
- **CDbl(Expression)**: Converts a numerical Expression to a Double data type
- **CInt(Expression)**: Converts a numerical Expression to an Integer data type (any fractional parts are rounded)
- **CLng(Expression)**: Converts a numerical Expression to a Long data type (any fractional parts are rounded)
- **CShort(Expression)**: Converts a numerical Expression to a Short data type (any fractional parts are rounded)
- **CSng(Expression)**: Converts a numerical Expression to a Single data type
- **CType(Expression, Type)**: Converts the Expression to the specified Type

We will use many of these conversion functions as we write code for our applications and examples. The Intellisense feature of the code window will usually direct us to the function we need.

3.5 Arithmetic Operators

Operators modify values of variables. The simplest operators carry out arithmetic operations. There are seven arithmetic operators in Visual Basic .NET.

**Addition** is done using the plus (+) sign and **subtraction** is done using the minus (-) sign. Simple examples are:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>7 + 2</td>
<td>9</td>
</tr>
<tr>
<td>Addition</td>
<td>3.4 + 8.1</td>
<td>11.5</td>
</tr>
<tr>
<td>Subtraction</td>
<td>6 - 4</td>
<td>2</td>
</tr>
<tr>
<td>Subtraction</td>
<td>11.1 – 7.6</td>
<td>3.5</td>
</tr>
</tbody>
</table>

**Multiplication** is done using the asterisk (*) and **division** is done using the slash (/). Simple examples are:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplication</td>
<td>8 * 4</td>
<td>32</td>
</tr>
<tr>
<td>Multiplication</td>
<td>2.3 * 12.2</td>
<td>28.06</td>
</tr>
<tr>
<td>Division</td>
<td>12 / 2</td>
<td>6</td>
</tr>
<tr>
<td>Division</td>
<td>45.26 / 6.2</td>
<td>7.3</td>
</tr>
</tbody>
</table>
The next operator is the **exponentiation** operator, represented by a carat symbol (^) or sometimes called a ‘hat.’ Some examples:

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 ^ 2</td>
<td>25</td>
</tr>
<tr>
<td>2.6 ^ 4</td>
<td>45.6976</td>
</tr>
<tr>
<td>3.7 ^ 3.1</td>
<td>57.733162065075</td>
</tr>
</tbody>
</table>

The other arithmetic operators are concerned with dividing integer numbers. The **integer division** operator is a backslash character (\). This works just like normal division except only integer (whole number) answers are possible - any fraction from the division is ignored. Conversely, the **modulus operator**, represented by the keyword `Mod`, divides two whole numbers, ignores the main part of the answer, and just gives you the remainder! It may not be obvious now, but the modulus operator is used a lot in computer programming. Examples of both of these operators are:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Division Result</th>
<th>Operation Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer division</td>
<td>7 \ 2</td>
<td>3.5</td>
<td>3</td>
</tr>
<tr>
<td>Integer division</td>
<td>23.2 \ 10</td>
<td>2.32</td>
<td>2</td>
</tr>
<tr>
<td>Integer division</td>
<td>25.2 \ 3.6</td>
<td>7.0</td>
<td>7</td>
</tr>
<tr>
<td>Modulus</td>
<td>7 Mod 4</td>
<td>1 Remainder 3</td>
<td>3</td>
</tr>
<tr>
<td>Modulus</td>
<td>14 Mod 3</td>
<td>4 Remainder 2</td>
<td>2</td>
</tr>
<tr>
<td>Modulus</td>
<td>25 Mod 5</td>
<td>5 Remainder 0</td>
<td>0</td>
</tr>
</tbody>
</table>

The mathematical operators have the following **precedence** indicating the order they are evaluated without specific groupings:

1. Exponentiation (^)
2. Multiplication (*) and division (/)
3. Integer division (\)
4. Modulus (Mod)
5. Addition (+) and subtraction (-)

If multiplications and divisions or additions and subtractions are in the same expression, they are performed in left-to-right order. **Parentheses** around expressions are used to force some desired precedence.

### 3.6 Comparison and Logical Operators

There are six comparison operators in Visual Basic .NET used to compare the value of two expressions (the expressions must be of the same data type). These are the basis for making decisions:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>=</td>
<td>Equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal to</td>
</tr>
</tbody>
</table>

It should be obvious that the result of a comparison operation is a Boolean value (**True** or **False**). **Examples:**

- \( A = 9.6, B = 8.1, A > B \) returns True
- \( A = 14, B = 14, A \geq B \) returns True
- \( A = “Visual”, B = "Visual", A = B \) returns True
- \( A = “Basic”, B = “Basic”, A <> B \) returns False
Logical operators operate on Boolean data types, providing a Boolean result. They are also used in decision making. We will use three logical operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not</td>
<td>Logical Not</td>
</tr>
<tr>
<td>And</td>
<td>Logical And</td>
</tr>
<tr>
<td>Or</td>
<td>Logical Or</td>
</tr>
</tbody>
</table>

3.7 Concatenation Operators

To concatenate two string data types (tie them together), use the & symbol or the + symbol, the string concatenation operators:

```
lblTime.Text = "The current time is" & Format(Now, "hh:mm")
txtSample.Text = “Hook this “+ “to this”
```

Visual Basic .NET offers other concatenation operators that perform an operation on a variable and assign the resulting value back to the variable. Hence, the operation

```
A = A + 1
```

Can be written using the addition concatenation operator (+=) as:

```
A += 1
```

This says A is incremented by 1.

Other concatenation operators and their symbols are:

<table>
<thead>
<tr>
<th>Operator Name</th>
<th>Operator Symbol</th>
<th>Operator Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>A &amp;= B</td>
<td>A = A &amp; B</td>
</tr>
<tr>
<td>String</td>
<td>A += B</td>
<td>A = A + B</td>
</tr>
<tr>
<td>Addition</td>
<td>A += B</td>
<td>A = A + B</td>
</tr>
<tr>
<td>Subtraction</td>
<td>A -= B</td>
<td>A = A – B</td>
</tr>
<tr>
<td>Multiplication</td>
<td>A *= B</td>
<td>A = A * B</td>
</tr>
<tr>
<td>Division</td>
<td>A /= B</td>
<td>A = A / B</td>
</tr>
<tr>
<td>Integer Division</td>
<td>A = B</td>
<td>A = A \ B</td>
</tr>
<tr>
<td>Exponentiation</td>
<td>A ^= B</td>
<td>A = A ^ B</td>
</tr>
</tbody>
</table>

Notice both &= and += can be used as the string concatenator.

3.8 Visual Basic .NET Functions

Visual Basic .NET offers a rich assortment of built-in functions that compute or provide various quantities. The on-line help utility will give you information on any or all of these functions and their use. The general form of a function is:

```
ReturnedValue = FunctionName(Arguments)
```

where Arguments represents a comma-delimited list of information needed by FunctionName to perform its computation. Once the arguments are supplied to the function it returns a value (ReturnedValue) for use in an application.
Some example functions are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Returned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asc</td>
<td>ASCII or ANSI code of a character</td>
</tr>
<tr>
<td>Chr</td>
<td>Character corresponding to a given ASCII or ANSI code</td>
</tr>
<tr>
<td>Format</td>
<td>Date or number converted to a text string</td>
</tr>
<tr>
<td>Instr</td>
<td>Locates a substring in another string</td>
</tr>
<tr>
<td>Len</td>
<td>Number of characters in a text string</td>
</tr>
<tr>
<td>Mid</td>
<td>Selected portion of a text string</td>
</tr>
<tr>
<td>Now</td>
<td>Current time and date</td>
</tr>
<tr>
<td>Str</td>
<td>Number converted to a text string</td>
</tr>
<tr>
<td>Timer</td>
<td>Number of seconds elapsed since midnight</td>
</tr>
<tr>
<td>Trim</td>
<td>Removes leading and trailing spaces from string</td>
</tr>
<tr>
<td>Val</td>
<td>Numeric value of a given text string</td>
</tr>
</tbody>
</table>

3.9 String Functions

Visual Basic .NET offers a powerful set of functions to work with string type variables, which are very important in Visual Basic .NET. The Text property of the label control and the text box control are string types. You will find you are constantly converting string types to numeric data types to do some math and then converting back to strings to display the information.

To convert a string type to a numeric value, use the Val function. As an example, to convert the Text property of a text box control named txtExample to a number, use:

```
Val(txtExample.Text)
```

This result can then be used with the various mathematical operators. The returned data type is Double. If you need the returned value to be some other data type, use one of the conversion functions seen earlier.

There are two ways to convert a numeric variable to a string. The Str function does the conversion with no regard for how the result is displayed. This bit of code can be used to display the numeric variable MyNumber in a text box control:

```
MyNumber = 3.14159
txtExample.Text = Str(MyNumber)
```

If you need to control the number of decimal points (or other display features), the Format function is used. This function has two arguments, the first is the number, the second a string specifying how to display the number (use on-line help to see how these display specifiers work). As an example, to display MyNumber with no more than two decimal points, use:

```
MyNumber = 3.14159
txtExample.Text = Format(MyNumber, "#.##")
```

In the display string ("#.##"), the pound signs represent place holders.

Many times, you need to extract substrings from string variables. The Mid function lets you extract a specified number of characters from anywhere in the string (you specify the string, the starting position and the number of characters to extract). This example extracts 6 characters from the string variable, starting at character 3:
MyString = “Visual Basic .NET is fun!”
MidString = Mid(MyString, 3, 6)

The MidString variable is equal to “sual B”

Perhaps, you just want a far left portion of a string. Use the Mid function with a starting position of 1. This example extracts the 3 left-most characters from a string:

MyString = “Visual Basic .NET is fun!”
LeftString = Mid(MyString, 1, 3)

The LeftString variable is equal to “Vis”

Getting the far right portion of a string with the Mid function is just a bit trickier. First, we need to introduce the Len function, which tells us the number of characters in (or length of) a string variable. For our example,

MyString = “Visual Basic .NET is fun!”
LenString = Len(MyString)

LenString will have a value of 25. Now, if we want the 6 far right characters in the string example, we use:

MyString = “Visual Basic .NET is fun!”
RightString = Mid(MyString, Len(MyString) – 6 + 1, 6)

The RightString variable is equal to “s fun!” Note that without the + 1 in the second argument, we would start one character too far to the left.

The Mid function can also be used to replace substrings in string variables, working in a reverse manner. Again, just specify the string, the starting position and number of characters to replace, but use the Mid function on the left side of the assignment operator. With our example, try:

MyString = “Visual Basic .NET is fun!”
Mid(MyString, 15, 3) = “net”

After this replacement, MyString will now have a value of:

MyString = “Visual Basic .net is fun!”

To locate a substring within a string variable, use the Instr function. Three arguments are used: starting position in String1 (optional), String1 (the variable), String2 (the substring to find). The function will work left-to-right and return the location of the first character of the substring (it will return 0 if the substring is not found). For our example:

MyString = “Visual Basic .NET is fun!”
Location = Instr(3, MyString, “sic”)

This says find the substring “sic” in MyString, starting at character 3 (if this argument is omitted, 1 is assumed). The returned Location will have a value of 10.

Related to the Instr function is the InstrRev function. This function also identifies substrings using identical arguments, but works right-to-left, or in reverse. So, with our example string:

MyString = “Visual Basic .NET is fun!”
Location = InstrRev(MyString, “is”)
This says find the substring “is” in MyString, starting at the right and working left. The returned Location will have a value of 20. Note if we use Instr instead, the returned Location would be 2.

Many times, you want to convert letters to upper case or vice versa. Visual Basic .NET provides two functions for this purpose: **UCase** and **LCase**. The UCase function will convert all letters in a string variable to upper case, while the LCase function will convert all letters to lower case. Any non-alphabetic characters are ignored in the conversion. And, if a letter is already in the desired case, it is left unmodified. For our example (modified a bit):

```vbnet
MyString = “Visual Basic .NET in 2002 is fun!”
A = UCase(MyString)
B = LCase(A)
```

The first conversion using UCase will result in:

```vbnet
A = “VISUAL BASIC .NET IN 2002 IS FUN!”
```

And the second conversion using LCase will yield:

```vbnet
B = “visual basic .net in 2002 is fun!”
```

Another useful pair of functions is the **Asc** and **Chr** functions. These work with individual characters. Every ‘type-able’ character has a numeric representation called an ASCII (pronounced “askey”) code. The Asc function returns the ASCII code for an individual character. For example:

```vbnet
Asc(“A”)
```

returns the ASCII code for the upper case A (65, by the way). The Chr function returns the character represented by an ASCII code. For example:

```vbnet
Chr(48)
```

returns the character represented by an ASCII value of 48 (a “1”). The Asc and Chr functions are used often in determining what a user is typing.

### 3.9 Random Number Object

In writing games and learning software, we use a random number generator to introduce unpredictability. This insures different results each time you try a program. Visual Basic .NET has a few ways to generate random numbers. We will use one of them – a random generator of integers. The generator uses the Visual Basic .NET Random object.

To use the Random object (named **MyRandom** here), it is first declared and created using:

```vbnet
Dim MyRandom As New Random()
```

This statement is placed with the variable declaration statements. Once created, when you need a random integer value, use the **Next** method of this Random object:

```vbnet
MyRandom.Next(Limit)
```

This statement generates a random integer value that is greater than or equal to 0 and less than **Limit**. Note it is less than limit, not equal to. For example, the method:
MyRandom.Next(5)

will generate random integers from 0 to 4. The possible values will be 0, 1, 2, 3 and 4.

As other examples, to roll a six-sided die, the number of spots would be computed using:

\[ \text{NumberSpots} = \text{MyRandom.Next}(6) + 1 \]

To randomly choose a number between 100 and 200, use:

\[ \text{Number} = \text{MyRandom.Next}(101) + 100 \]

### 3.10 Math Functions

A last set of functions we need are mathematical functions (yes, programming involves math!). Visual Basic .NET provides a set of functions that perform tasks such as square roots, trigonometric relationships, and exponential functions.

Each of the Visual Basic .NET math functions comes from the **Math** class of the .NET framework (an object-oriented programming concept). This means that each function name must be preceded by **Math.** (say Math-dot) to work properly. The functions and the returned values are:

<table>
<thead>
<tr>
<th>Math Function</th>
<th>Value Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.Abs</td>
<td>Returns the absolute value of a specified number</td>
</tr>
<tr>
<td>Math.E</td>
<td>A constant, the natural logarithm base</td>
</tr>
<tr>
<td>Math.Exp</td>
<td>Returns a Double value containing ( e ) (the base of natural logarithms) raised to the specified power</td>
</tr>
<tr>
<td>Math.Log</td>
<td>Returns a Double value containing the natural logarithm of a specified number</td>
</tr>
<tr>
<td>Math.Max</td>
<td>Returns the largest of two numbers</td>
</tr>
<tr>
<td>Math.Min</td>
<td>Returns the smaller of two numbers</td>
</tr>
<tr>
<td>Math.PI</td>
<td>A constant that specifies the ratio of the circumference of a circle to its diameter</td>
</tr>
<tr>
<td>Math.Round</td>
<td>Returns the number nearest the specified value</td>
</tr>
<tr>
<td>Math.Sin</td>
<td>Returns a Double value containing the sine of the specified angle</td>
</tr>
<tr>
<td>Math.Sqrt</td>
<td>Returns a Double value specifying the square root of a number</td>
</tr>
</tbody>
</table>

**Examples:**

- Math.Abs(-5.4) returns the absolute value of -5.4 (returns 5.4)
- Math.Cos(2.3) returns the cosine of an angle of 2.3 radians
- Math.Max(7, 10) returns the larger of the two numbers (returns 10)
- Math.Sqrt(4.5) returns the square root of 4.5
3.11 Visual Basic .NET Decisions - If Statements

The concept of an If statement for making a decision is very simple. We check to see if a particular condition is true. If so, we take a certain action. If not, we do something else. If statements are also called branching statements. Branching statements are used to cause certain actions within a program if a certain condition is met.

The simplest branching statement is the single line If/Then statement:

\[ \text{If Condition Then [Do This]} \]

In this statement, if Condition (some expression with a Boolean result) is True, then whatever single statement follow the Then keyword will be executed. If Condition is False, nothing is done and program execution continues following this line of code. Example:

\[ \text{If (Balance – Check) < 0 Then Trouble = True} \]

Here, if and only if Balance - Check is less than zero, the Trouble variable is set to True.

Single line If/Then statements are not considered to be ‘structured programming.’ They are sloppy programming practice. But, having said this, you will see that this author uses them (a bad habit I’m trying to phase out).

The structured approach to decisions is the If/Then/End If blocks which allow multiple statements to be processed for a decision:

\[ \text{If Condition Then} \]
\[ \quad \text{[process this code]} \]
\[ \text{End If} \]

Here, if Condition is True, the code bounded by the If/End If is executed. If Condition is False, nothing happens and code execution continues after the End If statement.

Example:

\[ \text{If Balance - Check < 0 Then} \]
\[ \quad \text{Trouble = True} \]
\[ \quad \text{lblBalance.ForeColor = Color.Red} \]
\[ \text{End If} \]

In this case, if Balance - Check is less than zero, two lines of information are processed: Trouble is set to True and the balance is displayed in red. Notice the indentation of the code between the If and End If lines. The Visual Basic .NET IntelliSense feature will automatically do this indentation. It makes understanding (and debugging) your code much easier. You can adjust the amount of indentation using the Options choice under the Tools menu. You will also notice that the Intellisense feature of the IDE will add an End If statement for you as soon as you type the If/Then line. What if you want to do one thing if a condition is True and another if it is False? Use an If/Then/Else/End If block:

\[ \text{If Condition Then} \]
\[ \quad \text{[process this code]} \]
\[ \text{Else} \]
\[ \quad \text{[process this code]} \]
\[ \text{End If} \]
In this block, if Condition is True, the code between the If and Else lines is executed. If Condition is False, the code between the Else and End If statements is processed.

Example:

If Balance - Check < 0 Then
    Trouble = True
    lblBalance.ForeColor = Color.Red
Else
    Trouble = False
    lblBalance.ForeColor = Color.Black
End If

Here, the same two lines are executed if you are overdrawn (Balance - Check < 0), but if you are not overdrawn (Else), the Trouble flag is turned off and your balance is in the black.

Finally, we can test multiple conditions by adding the ElseIf statement:

   If Condition1 Then
       [process this code]
   ElseIf Condition2 Then
       [process this code]
   ElseIf Condition3 Then
       [process this code]
   Else
       [process this code]
   End If

In this block, if Condition1 is True, the code between the If and first ElseIf line is executed. If Condition1 is False, Condition2 is checked. If Condition2 is True, the indicated code is executed. If Condition2 is not true, Condition3 is checked. Each subsequent condition in the structure is checked until a True condition is found, a Else statement is reached or the End If is reached.

Example:

If Balance - Check < 0 Then
    Trouble = True
    lblBalance.ForeColor = Color.Red
ElseIf Balance – Check = 0 Then
    Trouble = False
    lblBalance.ForeColor = Color.Yellow
Else
    Trouble = False
    lblBalance.ForeColor = Color.Black
End If
Now, one more condition is added. If your Balance equals the Check amount (**ElseIf** Balance - Check = 0), you’re still not in trouble and the balance is displayed in yellow.

In using branching statements, make sure you consider all viable possibilities in the If/Else/End If structure. Also, be aware that each If and ElseIf in a block is tested sequentially. The first time an If test is met, the code associated with that condition is executed and the If block is exited. If a later condition is also True, it will never be considered.

### 3.12 Select Case - Another Way to Branch

In addition to If/Then/Else type statements, the **Select Case** format can be used when there are multiple selection possibilities. Select Case is used to make decisions based on the value of a single variable. The structure is:

```
Select Case Variable
  Case [variable has this value]
    [process this code]
  Case [variable has this value]
    [process this code]
  Case [variable has this value]
    [process this code]
  Case Else
    [process this code]
End Select
```

The way this works is that the value of **Variable** is examined. Each **Case** statement is then sequentially examined until the value matches one of the specified cases. Once found, the corresponding code is executed. If no case match is found, the code in the Case Else segment (if there) is executed.

As an example, say we've written this code using the **If** statement:

```vbnet
If Age = 5 Then
  Category = "Five Year Old"
ElseIf Age >= 13 and Age <= 19 Then
  Category = "Teenager"
ElseIf (Age >= 20 and Age <= 35) Or Age = 50 Or (Age >= 60 and Age <= 65) Then
  Category = "Special Adult"
ElseIf Age > 65 Then
  Category = "Senior Citizen"
Else
  Category = "Everyone Else"
End If
```

This will work, but it is ugly code and difficult to maintain.
The corresponding code with **Select Case** is much 'cleaner':

```vbnet
Select Case Age
  Case 5
    Category = "Five Year Old"
  Case 13 To 19
    Category = "Teenager"
  Case 20 To 35, 50, 60 To 65
    Category = "Special Adult"
  Case Is > 65
    Category = "Senior Citizen"
  Case Else
    Category = "Everyone Else"
End Select
```

Like the If structure, the Select Case searches down each Case until it finds the first 'match.' Then, that code is executed. Only one Case for each Select Case can be executed. There is no possibility of a 'global' Select Case that would execute multiple Case statements. Make sure each Select Case you use accurately reflects the decision you are trying to implement.

### 3.13 Visual Basic .NET Looping

Many applications require repetition of certain code segments. For example, you may want to roll a die (simulated die of course) until it shows a six. Or, you might generate financial results until a certain sum of returns has been achieved. This idea of repeating code is called iteration or **looping**.

**Do/Loop**

In Visual Basic .NET looping is done with the **Do/Loop** format:

```vbnet
  Do
    [process this code]
  Loop
```

In this structure, all code between the **Do** (starts the loop) and the **Loop** statement is repeated until the loop is exited. This brings up a very important point – if you use a loop, make sure you can get out of the loop!! It is especially important in the event-driven environment of Visual Basic .NET. As long as your code is operating in some loop, no events can be processed. The **Exit Do** statement will get you out of a loop and transfer program control to the statement following the **Loop** statement. Of course, you need logic in a loop to decide when an Exit Do is appropriate.

Another way to exit a loop is to test conditions prior to or following execution of the code in loop. This is achieved using **Until** and **While** statements. These statements can be used with the Do statement or the Loop statement. Let's look at examples of all possible combinations.
A **Do While/Loop** structure:

```
Do While Condition
    [process this code]
Loop
```

In this structure, the loop is repeated ‘as long as’ the Boolean expression Condition is True. Note a Do While/Loop structure will not execute even once if the While condition is False the first time through. If we do enter the loop, it is assumed at some point Condition will become False to allow exiting.

**Example:**

```
Counter = 1
Do While Counter <= 1000
    Counter += 1
Loop
```

This loop repeats as long as (While) the variable Counter is less than or equal to 1000.

A **Do Until/Loop** structure:

```
Do Until Condition
    [process this code]
Loop
```

In this structure, the loop is repeated ‘until’ the Boolean expression Condition is True. Note a Do Until/Loop structure will not be entered if the Until condition is already True on the first encounter. However, once the loop is entered, it is assumed at some point Condition will become True to allow exiting.

**Example:**

```
Rolls = 0
Counter = 0
Do Until Counter = 10
    'Roll a simulated die
    Roll += 1
    If MyRandom.Next(6) + 1 = 6 Then
        Counter += 1
    End If
Loop
```

This loop repeats Until the Counter variable equals 10. The Counter variable is incremented each time a simulated die rolls a 6. The Roll variable tells you how many rolls of the die were needed.

A **Do/Loop While** structure:

```
Do
    [process this code]
Loop While Condition
```

This loop repeats as long as the Boolean expression Condition is true. The loop is always executed at least once. Somewhere in the loop, Condition must be changed to False to allow exiting.

**Example:**
Sum = 0
Do
    Sum += 3
Loop While Sum <= 50

In this example, we increment a sum by 3 until that sum exceeds 50 (or **While** the sum is less than or equal to 50).

A **Do/Loop Until** structure:

Do
    [process this code]
Loop Until Condition

This loop repeats until the Boolean expression **Condition** is True. The loop is always executed at least once. Somewhere in the loop, **Condition** must be become True for proper exiting.

**Example:**

Sum = 0
Counter = 0
Do
    'Roll a simulated die
    Sum += MyRandom.Next(6) + 1
    Counter += 1
Loop Until Sum > 30

This loop rolls a simulated die **Until** the Sum of the rolls exceeds 30. It also keeps track of the number of rolls (Counter) needed to achieve this sum.

Again, make sure you can always get out of a loop! Infinite loops are never nice. If you get into one, try **Ctrl+Break**. That sometimes works - other times the only way out is rebooting your machine!

### 3.14 Visual Basic .NET Counting

With **Do/Loop** structures, we usually didn’t know, ahead of time, how many times we execute a loop or iterate. If you know how many times you need to iterate on some code, you want to use **Visual Basic .NET counting**. Counting is useful for adding items to a list or perhaps summing a known number of values to find an average.

Visual Basic .NET counting is accomplished using the **For/Next** loop:

**For** Variable = Start **To** End [**Step** Increment]
    [process this code]
**Next** Variable

In this loop, **Variable** is the counter (doesn’t necessarily need to be a whole number). The first time through the loop, **Variable** is initialized at **Start**. Each time the corresponding **Next** statement is reached, **Variable** is incremented by an amount **Increment**. If the **Step** value is omitted, a default increment value of one is used. Negative increments are also possible. The counting repeats until **Variable** equals or exceeds the final value **End**.
Example:

For Degrees = 0 To 360 Step 10
  ‘convert to radians
  R = Degrees * Math.PI / 180
  A = Math.Sin(R)
  B = Math.Cos(R)
  C = Math.Tan(R)
Next Degrees

In this example, we compute trigonometric functions for angles from 0 to 360 degrees in increments (steps) of 10 degrees. Notice that the Intellisense feature of the IDE will add a Next statement for you as soon as you type the For line. I suggest always appending the variable name to the Next statement as soon as it appears. This is not necessary (the variable name in the Next statement is optional), but good programming practice to make your code clearer.

Another Example:

For Countdown = 10 to 0 Step –1
  lblTimeRemaining.Text = Format (Countdown, “0”) + “Seconds”
Next Countdown

NASA called and asked us to format a label control to countdown from 10 to 0. The loop above accomplishes the task.

And, Another Example:

Dim MyValues(100) as Double
Sum = 0
For I = 1 to 100
  Sum += MyValues(I)
Next I
Average = Sum / 100

This code finds the average value of 100 numbers stored in the array MyValues. It first sums each of the values in a For/Next loop. That sum is then divided by the number of terms (100) to yield the average.

You may exit a For/Next loop early using an Exit For statement. This will transfer program control to the statement following the Next statement.
3.15 Exercises

Exercise 3-1 - Random Number Problem

Build an application where each time a button is clicked, a random number from 1 to 100 is displayed.

Exercise 3-2 - Price Problem

The neighborhood children built a lemonade stand. The hotter it is, the more they can charge. Build an application that produces the selling price, based on temperature:

<table>
<thead>
<tr>
<th>Temperature</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;50</td>
<td>Don’t bother</td>
</tr>
<tr>
<td>50 – 60</td>
<td>20 Cents</td>
</tr>
<tr>
<td>61 – 70</td>
<td>25 Cents</td>
</tr>
<tr>
<td>71 – 80</td>
<td>30 Cents</td>
</tr>
<tr>
<td>81 – 85</td>
<td>40 Cents</td>
</tr>
<tr>
<td>86 – 90</td>
<td>50 Cents</td>
</tr>
<tr>
<td>91 – 95</td>
<td>55 Cents</td>
</tr>
<tr>
<td>96 – 100</td>
<td>65 Cents</td>
</tr>
<tr>
<td>&gt;100</td>
<td>75 Cents</td>
</tr>
</tbody>
</table>

Exercise 3-3 - Odd Integers Problem

Build an application that adds consecutive odd integers (starting with one) until the sum exceeds a target value. Display the sum and how many integers were added.

Exercise 3-4 - Pennies Problem

Here’s an old problem. Today, I'll give you a penny. Tomorrow, I'll give you two pennies. I'll keep doubling the amount I'll give you for 30 days. How much will you have at the end of the month (better use a Long integer type to keep track)?

Exercise 3-5 - Code Problem

Build an application with a text box and two buttons. Type a word or words in the text box. Click one of the buttons. Subtract one from the ASCII code of each character in the typed word(s), then redisplay it. This is a simple encoding technique. When you click the other button, reverse the process to decode the word.
CHAPTER FOUR

Object-Oriented concepts

4.1 Introduction

The object-oriented terms class and object often create some confusion because they can be easily misused by developers when describing a system.

4.1.1 Class

A class is a template or blueprint that defines an object’s attributes and operations and that is created at design time. A class is an abstract data type containing data, a set of functions to access and manipulate the data, and a set of access restrictions on the data and on the functions.

You can think of a class as a template or a blueprint for an object. This blueprint defines attributes for storing data and defines operations for manipulating that data. A class also defines a set of restrictions to allow or deny access to its attributes and operations.

A car is an example of a class. We know that a car has attributes, such as the number of wheels, the color, the make, the model, and so on. We know that it also has operations, including unlock door, open door, and start engine.

4.1.2 Object

An object is a running instance of a class that consumes memory and has a finite lifespan. Objects are instances of classes. A single blueprint or class can be used as a basis for creating many individual and unique objects.

If you consider classes and objects in Visual Basic terms, a class is created at design time and will exist forever, whereas an object is instantiated at run time and will only exist as long as required during the application execution. For example, an instance of a car would contain specific information for each attribute, such as number or wheels equals four, color equals blue, and so on.

Objects exhibit three characteristics:

- Identity
- Behavior
- State

It may be useful to discuss the differences between objects and classes with another example that is relevant to Visual Basic developers. A button is defined internally in Visual Basic as being rectangular, as having a caption, and as having the ability to be clicked. This is an example of a class.

You can place as many instances of a button on a form as you want, and changing the caption of one doesn’t affect the others. These are examples of objects. Likewise, clicking one instance of a button does not cause the others to be clicked.

4.1.2.1 Identity

One object must be distinguishable from another object of the same class. Without this characteristic, it would be impossible to tell the difference between the two objects, and this would cause great difficulties for developers. This difference could be a simple identifier such as a unique ID number assigned to each object, or several of each object’s attributes could be different from those of the other objects.
A particular car could be the same make, model, and color as another car, but the registration numbers cannot be identical. This difference provides a way to distinguish two otherwise identical cars.

4.1.2.2 Behavior

Objects exist to provide a specific behavior that is useful. If they did not exhibit this characteristic, we would have no reason to use them. The main behavior or purpose of a car is to transport people from one location to another. If the car did not provide this behavior, it would not perform a useful function.

4.1.2.3 State

State refers to the attributes or information that an object stores. These attributes are often manipulated by an object’s operations. The object’s state can change by direct manipulation of an attribute, or as the result of an operation. A well-designed object often only allows access to its state by means of operations because this limits incorrect setting of the data.

A car keeps track of how far it has traveled since it was created in the factory. This data is stored internally and can be viewed by the driver. The only way to alter this data is to drive the car, which is an operation that acts upon the internal state.

4.2 Encapsulation

How an Object Performs Its Duties Is Hidden from the Outside World, Simplifying Client Development. Clients can call a method of an object without understanding the inner workings or complexity. Any changes made to the inner workings are hidden from clients Encapsulation is the process of hiding the details about how an object performs its duties when asked to perform those duties by a client. This has some major benefits for designing client applications:

- Client development is simplified because the clients can call a method or attribute of an object without understanding the inner workings of the object.
- Any changes made to the inner workings of the object will be invisible to the client.
- Because private information is hidden from the client, access is only available by means of appropriate operations that ensure correct modification of data.

Example

Driving a car is an example of encapsulation. You know that when you press the accelerator the car will move faster. You do not need to know that the pedal increases the amount of fuel being fed into the engine, producing more fuel ignition and thereby speeding up the output to the axle, which in turn speeds up the car’s wheels, which has the final effect of increasing your speed. You simply need to know which pedal to press to have the desired effect.

Likewise, if the car manufacturer changes the amount of fuel being mixed with oxygen to alter the combustion, or creates a drive-by-wire accelerator pedal, you do not need to know in order to increase your speed. However, if the manufacturer replaces the accelerator pedal with a sliding throttle device, similar to what you would find in an aircraft, you may need to know about it!
4.3 Abstraction

Abstraction is selective ignorance. It involves the following:

- Decide what is important and what is not
- Focus on and depend on what is important
- Ignore and do not depend on what is unimportant
- Use encapsulation to enforce an abstraction

Abstraction is the practice of focusing only on the essential aspects of an object. It allows you to selectively ignore aspects that you deem unimportant to the functionality provided by the object. A good abstraction only provides as many operations and attributes as are required to get the job done. The more operations and attributes provided, the more difficult to use the object becomes.

If an object is simple to use because it includes only the essential operations, there is a greater possibility that it can be reused by other applications. A good abstract design will also limit a client's dependency on a particular class. If a client is too dependent on the way an operation is performed by an object, any modification to the internal aspects of that operation may impact the client, requiring that additional work be completed. This is often known as the principle of minimal dependency.

4.4 Attributes and Operations

Attributes are the data contained in a class. Operations are the actions performed on that data. Accessibility: Public (+), Private (-), Protected (#). Classes are usually made up of data and actions performed on this data. These are known as attributes and operations respectively, but developers also call them properties and methods. These attributes and operations are also defined with an accessibility setting.

4.4.1 Attributes

Attributes are the data members of the class. They can be of any data type, including String, Decimal, or even another class. Each attribute will also have an accessibility option specified, as shown in the following table. In Visual Basic .NET, public attributes will be implemented as either class-level variables or, more appropriately, as class properties that encapsulate internal variables.

4.4.2 Operations

Operations are the actions performed on internal data within the class. They can take parameters, return values, and have different accessibility options specified in the same way that attributes can. In Visual Basic .NET, these are implemented as either functions or subroutines.

4.4.3 Accessibility

Attributes and operations can be defined with one of the access modifiers as indicated below:

**Value Meaning**
- Public (+) - Accessible to the class itself and to any client of the class.
- Protected(#) - Only accessible to a child class when used for inheritance.
- Private (-) - Only accessible by code within the class that defines the private attribute or operation.
4.5 Inheritance

Inheritance specifies an “is-a-kind-of” relationship. Multiple classes share the same attributes and operations, allowing efficient code reuse.

Examples:

- A customer “is a kind of” person
- An employee “is a kind of” person

Inheritance is the concept of reusing common attributes and operations from a base class in a derived class. If the base class does not contain implementation code and will never be instantiated as an object, it is known as an abstract class.

The example below shows an inheritance relationship between the Customer, Employee, and Person classes. The Person superclass has attributes defined as Name, Gender, and Date of Birth and contains the operations Create, Remove, and Validate. These are all attributes and operations that could equally be applied to a Customer or Employee subclass, providing a good deal of reuse. Specialization occurs when a subclass is created and includes attributes or operations specific to that class.

The Customer class has the extra attributes CustomerID, CreditRating, and LastContacted in addition to the inherited ones from the Person superclass. It also defines its own operation named AddOrder that is specific to the Customer class. Having an operation called AddOrder would not make sense for either the Person class or the Employee class.

The Employee class has the extra attributes EmployeeID, Department, and Manager. It also defines a unique operation named Pay that would not be required in either the Person superclass or the Customer subclass.

If a superclass is not an abstract class and contains some implementation code, the subclass can inherit the code from the superclass or override it by providing its own code. This reuse of code is known as implementation inheritance and is the most powerful form of reuse.

Although implementation inheritance is very useful, it can lead to class diagrams and code that are complex and difficult to read. You should ensure that implementation inheritance is used appropriately and not overused.

4.6 Interfaces

Interfaces only define the method signatures. Classes define the implementation of the code for the Interface methods. Interface inheritance means only the Interface is inherited, not the implementation code.

Interfaces are similar to abstract classes. They define the method signatures used by other classes but do not implement any code themselves. Interface inheritance means that only the method signatures are inherited and that any implementation code is not. You would need to create separate code in the appropriate inherited method of each derived class to achieve any required functionality.

Reuse is therefore more limited in interface inheritance as compared to implementation inheritance because you must write code in multiple locations.
4.7 Polymorphism

The same operation behaves differently when applied to objects based on different classes. Often based on Interface inheritance

- Classes inherit from interface base class
- Each derived class implements its own version of code
- Clients can treat all objects as if they are instances of the base class, without knowledge of the derived classes

Polymorphism is the ability to call the same method on multiple objects that have been instantiated from different subclasses and generate differing behavior. This is often achieved by using interface inheritance. If two subclasses inherit the same interface, each of them will contain the same method signatures as the superclass. Each one will implement the code in an individual way, allowing different behavior to be created from the same method.

In the above example, the **Customer** and **Employee** classes have inherited from the **Person** superclass. Each class implements its own version of the **Create** method differently, but, because they both inherit the same interface, a client could treat both classes the same.
CHAPTER FIVE

Object Oriented Programming in Visual Basic.NET

VB.NET not only provides us with new OO features, but it also changes the way we implement some of the features we are used to from VB6. As we go through these features we’ll cover both the new capabilities and also explore the changes to existing features.

5.1 Creating Classes

Unlike in VB6, VB.NET allows us to put more than one class in a single source file. While we don’t have to take this approach, it can be nice since we can reduce the overall number of files in a project – possibly making it more maintainable.

Additionally, VB.NET provides support for the concept of .NET namespaces. There are also changes to the syntax used to create Property methods, and we can overload methods in our classes. We’ll look at all these features shortly. First though, let’s look at how we add a class to a project.

In order to do this we need to create a new Windows Application project and choose the Project | Add Class menu option to bring up the Add New Item dialog:

This is the common dialog used for adding any type of item to our project – in this case it defaults to adding a class module. Regardless of which type of VB source file we choose (form, class, module, etc.) we’ll end up with a file ending in a .vb extension.

It is the content of the file that determines its type, not the file extension. The IDE creates different starting code within the file based on the type we choose.

We can name the class TheClass in this dialog and, when we click Open, a new file will be added to our project, containing very simple code:

```
Public Class TheClass
End Class
```

Though a .vb file can contain multiple classes, modules and other code, the normal behavior from the IDE is the same as we’ve had in VB since its inception – one class, module, or form per file. We can manually add other code to the files created by the IDE with no problems, but when we ask the IDE to create a class for us it will always do so by adding a new file to the project.

5.1.1 Class Keyword

As shown in this example, we now have a Class keyword along with the corresponding End Class. This new keyword is needed in order for a single source file to contain more than one class. Any time we want to create a class in VB.NET, we simply put all the code for the class within the Class…End Class block. For instance:
Within a given source file (any .vb file) we can have many of these: Class...End Class blocks, one after another.

### 5.1.2 Classes and Namespaces

Namespaces are central to the .NET environment, as they provide a mechanism by which classes can be organized into logical groupings, making them easier to find and manage.

Namespaces in VB.NET are declared using a block structure. For example:

```vbnet
Namespace TheNamespace
    Public Class TheClass
    End Class
End Namespace
```

Any classes, structures, or other types declared within the `Namespace...End Namespace` block will be addressed using that namespace. In this example, our class is referenced using the namespace, so declaring a variable would be done as follows:

```vbnet
Private obj As TheNamespace.TheClass
```

Because namespaces are created using a block structure, it is possible for a single source file to contain not only many classes, but also many namespaces.

Also, classes within the same namespace can be created in separate files. In other words, within a VB.NET project we can use the same namespace in more than one source file – and all the classes within those namespace blocks will be part of that same namespace.

For instance, if we have one source file with the following code:

```vbnet
Namespace TheNamespace
    Public Class TheClass
    End Class
End Namespace
```

And we have a separate source file in the project with the following code:

```vbnet
Namespace TheNamespace
    Public Class TheOtherClass
    End Class
End Namespace
```

Then we’ll have a single namespace – `TheNamespace` – with two classes – `TheClass` and `TheOtherClass`. 
It is also important to remember that VB.NET projects, by default, have a root namespace that is part of the project's properties. By default this root namespace will have the same name as our project. When we use the `Namespace` block structure, we are actually adding to that root namespace. So, in our example, if the project is named `MyProject`, then we could declare a variable as:

```
Private obj As MyProject.TheNamespace.TheClass
```

In order to change the root namespace, use the Project | Properties menu option. The root namespace can be cleared as well, meaning that all `Namespace` blocks become the root level for the code they contain.

### 5.1.3 Creating Methods

Methods in VB.NET are created just like they are in VB6 – using the `Sub` or `Function` keywords. A method created with `Sub` does not return a value, while a `Function` must return a value as a result.

```
Sub DoSomething()
End Sub

Function GetValue() As Integer
End Function
```

We retain the three scoping keywords we are used to, and have one more:

- **Private** – callable only by code within our class
- **Friend** – callable only by code within our project/component
- **Public** – callable by code outside our class
- **Protected** – callable only by code within our subclasses.
- **Protected Friend** – callable only by code within our project/component and by code in our subclasses.

Parameters to methods are now declared `ByVal` by default, rather than `ByRef`. We can still override the default behavior through explicit use of the `ByRef` keyword.

### 5.1.4 Creating Properties

In Chapter 3 we discussed the changes to the way `Property` routines are created. In the past we’d create separate routines for `Property Get` and `Property Set`. Now these are combined into a single structure:

```
Private mstrName As String
Public Property Name() As String
    Get
        Return mstrName
    End Get
    Set
        mstrName = Value
    End Set
End Property
```
5.1.5 New Method

Like the situation with `Sub Main`, `Class_Initialize` is called before any other code in a VB6 class. Again, it is called before the error handling mechanism is fully in place, making debugging very hard; errors show up at the client as a generic failure to instantiate the object. Additionally, `Class_Initialize` accepts no parameters – meaning there is no way in VB6 to initialize an object with data as it is created.

VB.NET eliminates `Class_Initialize` in favor of full-blown constructor methods, which have full error handling capabilities and do accept parameters. This means we can initialize our objects as we create them – a very important and powerful feature. The constructor method in VB.NET is `Sub New`. The simplest constructor method for a class is one that accepts no parameters – quite comparable to `Class_Initialize`:

```vbnet
Public Class TheClass
    Public Sub New()
        ' initialize object here
    End Sub
End Class
```

With this type of constructor, creating an instance of our class is done as follows:

```vbnet
Dim obj As New TheClass()
```

This example is directly analogous to creating a VB6 object with code in `Class_Initialize`.

However, more often than not we’d prefer to actually initialize our object with data as it is created. Perhaps we want to have the object load some data from a database, or perhaps we want to provide it with the data directly. Either way, we want to provide some data to the object as it is being created.

This is done by adding a parameter list to the `New` method:

```vbnet
Public Class TheClass
    Public Sub New(ByVal ID As Integer)
        ' use the ID value to initialize the object
    End Sub
End Class
```

Now, when we go to create an instance of the class, we can provide data to the object:

```vbnet
Dim obj As New TheClass(42)
```

To increase flexibility we might want to optionally accept the parameter value. This can be done in two ways – through the use of the `Optional` keyword to declare an optional parameter, or by overloading the `New` method. To use the `Optional` keyword, we simply declare the parameter as optional:

```vbnet
Public Sub New(Optional ByVal ID As Integer = -1)
    If ID = -1 Then
        ' initialize object here
    Else
        ' use the ID value to initialize the object
    End If
End Sub
```
This approach is far from ideal, however, since we have to check to see if the parameter was or wasn’t provided, and then decide how to initialize the object. It would be clearer to just have two separate implementations of the New method – one for each type of behavior. This is accomplished through overloading:

```visualbasic
Public Overloads Sub New()
    ' initialize object here
End Sub
Public Overloads Sub New(ByVal ID As Integer)
    ' use the ID value to initialize the object
End Sub
```

Not only does this approach avoid the conditional check and simplify our code, but it also makes the use of our object clearer to any client code. The overloaded New method is shown by IntelliSense in the VS.NET IDE, making it clear that New can be called both with and without a parameter.

In fact, through overloading we can create many different constructors if needed – allowing our object to be initialized in a number of different ways.

Constructor methods are optional in VB.NET. The only exception being when we’re using inheritance and the parent class has only constructors that require parameters.

### 5.2 Inheritance

While the OO features of VB have been very powerful and useful, we have been held back in many cases by the lack of inheritance in the language. Inheritance is the ability of a class to gain the interface and behaviors of an existing class. The process by which this is accomplished is called sub-classing. When we create a new class that inherits the interface and behaviors from an existing class, we have created a subclass of the original class. This is also known as an “is-a” relationship, where the new class “is-a” type of original class.

There is a lot of terminology surrounding inheritance – much of it redundant. The original class, from which we inherit interface and behavior, is known by the following interchangeable terms:

- Parent class
- Superclass
- Base class

The new class that inherits the interface and behaviors is known by the following interchangeable terms:

- Child class
- Subclass

Inheritance is also sometimes called **generalization**. In fact this is the term used within the Universal Modeling Language (UML) – the most commonly used object diagramming notation.

Inheritance is often viewed through the lens of biology, where, for example, a dog is a canine and a canine is a mammal. Hence, by being a canine, a dog inherits all the attributes and behavior of a mammal. While useful for visualization, these analogies only go so far.

VB.NET does not allow multiple inheritance – where a subclass is created by inheriting from more than one parent class. This feature is not supported by the .NET runtime and thus is not available from VB.NET. VB.NET does allow deep inheritance hierarchies where a class is sub-classed from a class that is sub-classed, but it doesn’t allow a class to be sub-classed from multiple parent classes all at once.
We can contrast inheritance, an “is-a” relationship, with another type of parent-child relationship – the “has-a” relationship. This is also known as aggregation or containment.

In a “has-a” relationship, the parent object owns one or more of the child objects, but the child objects are of different types from the parent. For instance, an Invoice has-a LineItem. The LineItem object isn't sub-classed from Invoice – it is an entirely different class that just happens to be owned by the Invoice parent.

This distinction is important, because the terms parent and child are used frequently when working with objects – sometimes when referring to inheritance and sometimes when referring to aggregation. It is important to understand which is which or things can get very confusing.

Within this section, we'll use the terms parent, child, and subclass – all in the context of inheritance.

5.2.1 Implementing Basic Inheritance

To explore inheritance, consider a business example with a sales order that has line items. We might have product line items and service line items. Both are examples of line items, but both are somewhat different as well. While we could certainly implement ProductLine and ServiceLine classes separately, they’d have a lot of common code between them. Redundant code is hard to maintain, so it would be nicer if they could somehow directly share the common code between them.

This is where inheritance comes into play. Using inheritance, we can create a LineItem class that contains all the code common to any sort of line item. Then we can create ProductLine and ServiceLine classes that inherit from LineItem – thus automatically gaining all the common code – including interface and implementation in an OO form.

A simple LineItem class might appear as:

```vbnet
Public Class LineItem
    Private mintID As Integer
    Private mstrItem As String
    Private msngPrice As Single
    Private mintQuantity As Integer
    Public Property ID() As Integer
        Get
            Return mintID
        End Get
        Set
            mintID = value
        End Set
    End Property
    Public Property Item() As String
        Get
            Return mstrItem
        End Get
        Set
            mstrItem = Value
        End Set
    End Property
    Public Property Price() As Single
        Get
            Return msngPrice
        End Get
        Set
    End Property
End Class
```
Public Property msngPrice() As Single
   Get
       Return msngPrice
   End Get
   Set
       msngPrice = Value
   End Set
End Property

Public Property Quantity() As Integer
   Get
       Return mintQuantity
   End Get
   Set
       mintQuantity = Value
   End Set
End Property

Public Function Amount() As Single
   Return mintQuantity * msngPrice
End Function
End Class

This class has things common to any line item – some basic data fields and a method to calculate the cost of the item.

If a line item is for a product, however, we might have additional requirements. The Item value should probably be validated to make sure it refers to a real product, and perhaps we want to provide a product description as well:

Public Class ProductLine
   Inherits LineItem
   Private mstrDescription As String
   Public ReadOnly Property Description() As String
      Get
         Return mstrDescription
      End Get
   End Property
   Public Sub New(ByVal ProductID As String)
      Item = ProductID
      ' load product data from database
      mstrDescription = “Test product description”
   End Sub
End Class

Note the use of the Inherits statement.

Inherits LineItem

It is this statement that causes the ProductLine class to gain all the interface elements and behaviors from the LineItem class. This means that we can have client code like this:

Protected Sub Button1_Click(ByVal sender As Object, ByVal e As System.EventArgs)
   Dim pl As ProductLine
   pl = New ProductLine(“123abc”)
   MessageBox.Show(pl.Item)
   MessageBox.Show(pl.Description)
End Sub
This code makes use of both the Item property (from the LineItem class) and the Description property from the ProductLine class. Both are equally part of the ProductLine class, since it is a subclass of LineItem.

Likewise, a line item for a service might have a date for when the service was provided, but otherwise be the same as any other line item:

```vbnet
Public Class ServiceLine
    Inherits LineItem
    Private mdtDateProvided As Date
    Public Sub New()
        Quantity = 1
    End Sub
    Public Property DateProvided() As Date
        Get
            Return mdtDateProvided
        End Get
        Set
            mdtDateProvided = Value
        End Set
    End Property
End Class
```

Again, notice the use of the Inherits statement that indicates this is a subclass of the LineItem class. The DateProvided property is simply added to the interface gained from the LineItem class.

### 5.2.2 Preventing Inheritance

By default any class we create can be used as a base class from which other classes can be created. There are times when we might want to create a class that cannot be subclassed. To do this we can use the NotInheritable keyword in our class declaration:

```vbnet
Public NotInheritable Class ProductLine
End Class
```

When this keyword is used, no other code may use the Inherits keyword to create a subclass of our class.

### 5.2.3 Inheritance and Scoping

When we create a subclass through inheritance, the new class gains all the Public and Friend methods, properties, and variables from the original class. Anything declared as Private in the original class will not be directly available to our code in the new subclass.

The exception to this is the New method. Constructor methods must be re-implemented in each subclass. We’ll discuss this in more detail later in the chapter.

For instance, we might rewrite the Amount methods from the LineItem class slightly:

```vbnet
Public Function Amount() As Single
    Return CalcAmount
End Function
Private Function CalcAmount() As Single
    Return fQuantity * fPrice
End Function
```
With this change, we can see that the `Public` method `Amount` makes use of a `Private` method to do its work.

When we subclass `LineItem` to create the `ServiceLine` class, any `ServiceLine` object will have an `Amount` method because it is declared as `Public` in the base class. The `CalcAmount` method, on the other hand, is declared as `Private` and so neither the `ServiceLine` class nor any client code will have any access to it.

Does this mean the `Amount` method will break when called through the `ServiceLine` object? Not at all. Since the `Amount` method’s code resides in the `LineItem` class, it has access to the `CalcAmount` method even though the `ServiceLine` class can’t see the method.

For instance, in our client code we might have something like this:

```vbnet
Protected Sub Button1_Click(ByVal sender As Object, ByVal e As System.EventArgs)
    Dim sl As ServiceLine
    sl = New ServiceLine()
    sl.Item = “delivery”
    sl.Price = 20
    sl.DateProvided = Now
    MsgBox(sl.Amount, MsgBoxStyle.Information, “Amount”)
End Sub
```

The result is displayed in a message box, thus illustrating that the `CalcAmount` method was called on our behalf even though neither our client code, nor the `ServiceLine` code directly made the call.

### 5.2.4 Protected Methods

Sometimes `Public` and `Private` aren’t enough. If we declare something as `Private` it is totally restricted to our class, while if we declare something as `Public` (or `Friend`) it is available to both subclasses and client code. There are times when it would be nice to create a method that is available to subclasses, but not to client code.

This is where the `protected` scope comes into play. When something is declared as `protected`, it is not available to any code outside of the class. However, it is available to classes that are derived from our class through inheritance.

For example:

```vbnet
Public Class ParentClass
    Protected TheValue As Integer
End Class
Public Class SubClass
    Inherits ParentClass
    Public Function GetValue() As Integer
        Return TheValue
    End Function
End Class
```

Here we have a parent class with a `protected` member – `TheValue`. This variable is not available to any client code. However, the variable is fully available to any code within `SubClass`, because it inherits from the parent.

In this example, `SubClass` has a `Public` method that actually does return the protected value – but the variable `TheValue` is not directly available to any client code (that is, code outside the class).
5.2.5 Overriding Methods

One key attribute of inheritance is that a subclass not only gains the behaviors of the original class, but it can also override those behaviors. We’ve already seen how a subclass can extend the original class by adding new Public, Protected, and Friend methods. However, by using the concept of overriding, a subclass can alter the behaviors of methods that were declared on the parent class.

By default, methods cannot be overridden by a subclass. To allow them to be overridden, the parent class must declare the method using the Overridable keyword:

```vbnet
Public Class Parent
    Public Overridable Sub DoSomething()
        MessageBox.Show("Hello from Parent")
    End Sub
End Class
```

We can also explicitly disallow overriding through the use of the NotOverridable keyword. Of course since this is the default, this keyword is rarely used.

However, it may be a good practice to explicitly define whether a method can or cannot be overridden to increase the clarity of code and to protect against the possibility that the default behavior might someday change.

If we then create a subclass, we can optionally override the behavior of DoSomething by using theOverrides keyword:

```vbnet
Public Class SubClass
    Inherits Parent
    Public Overrides Sub DoSomething()
        MessageBox.Show("Hello from SubClass")
    End Sub
End Class
```

Now we can write client code such as:

```vbnet
Dim obj As New SubClass()  
obj.DoSomething()           
```

The result will be a message dialog containing the text Hello from SubClass. This isn’t surprising – after all, we overrode the DoSomething method with our new code.

5.2.6 Virtual Methods

However, consider the following client code:

```vbnet
Dim obj As Parent
    obj = New SubClass()
    obj.DoSomething()  
```

First off, it seems odd to declare a variable of type Parent, but then create a Subclass object instead. This is perfectly acceptable however – it is yet another way of implementing polymorphism. Since Subclass "is-a" Parent, any Parent or Subclass variable can hold a reference to a Subclass object.
This is true in general. When using inheritance, a variable of the parent type can always hold references to any child type created from the parent.

What may be more surprising is the message that is displayed in our message box when this code is run. The message we see is ‘Hello’ from Subclass.

How can this be? The variable is declared as type Parent – shouldn’t the Parent implementation be called? The reason the DoSomething implementation from the child class is called is that the method is virtual. The concept of a virtual method is such that the “bottom-most” implementation of the method is always used in favor of the parent implementation – regardless of the data type of the variable being used in the client code.

Unlike many object-oriented languages, all methods in VB.NET are virtual.

If we create an object from type Parent with the following code:

```
Dim obj As Parent
obj = New Parent()
obj.DoSomething()
```

The DoSomething implementation in the Parent class will be invoked since that is the type of object we created as shown in the following diagram:

We can also create the object from the SubSubClass class:

```
Dim obj As Parent
obj = New SubSubClass()
obj.DoSomething()
```

In this case, the class doesn’t directly implement DoSomething, so we start looking back up the inheritance chain:

The first class up that chain is SubClass, which does have an implementation – so it is that implementation which is invoked. No code from the Parent class is invoked at all.

5.2.7 Me Keyword

The Me keyword is used any time we want our code to refer to methods within the current object. This was used in VB6 – when we might have utilized the Me keyword to refer to the current form, or the current instance of an object – and the same is true in VB.NET.

The Me keyword is analogous to the this keyword in C++ and C# languages.

The Me keyword is usually optional, since any method call is assumed to refer to the current object unless explicitly noted otherwise. The exception is when we are working with shadowed variables.

A shadowed variable is a procedure-level variable with the same name as a class-level variable. For instance:

```vbnet
Public Class TheClass
    Private strName As String
    Public Sub DoSomething()
        Dim strName As String
        strName = “Fred”
    End Sub
End Class
```
The variable *strName* is declared at the class level and within the *DoSomething* method. Within that method only the local, or shadowed, variable is used unless we explicitly reference the class-level variable with the *Me* keyword:

```vbnet
Public Sub DoSomething()
    Dim strName As String
    strName = "Fred"  ' sets the local variable’s value
    Me.strName = "Mary"  ' sets the class level variable’s value
End Sub
```

Here we can see that *strName* can be used to reference the local variable, while *Me.strName* can be used to reference the class-level variable.

As useful as the *Me* keyword can be for referring to the current object, when we start working with inheritance, it isn’t enough to do everything we want.

There are two issues we need to deal with. Sometimes we may want to explicitly call into our parent class – and sometimes we may want to ensure that the code in our class is being called, rather than the code in some subclass that has inherited our code.

### 5.2.8 MyBase Keyword

At times we might want to explicitly call into methods in our parent class. Remember that when we override a method it is virtual – so the method call will invoke the “bottom-most” implementation – not the parent implementation. However, sometimes we might need that parent implementation.

To invoke the parent class from within our subclass we can use the *MyBase* keyword. For instance:

```vbnet
Public Class SubClass
    Inherits Parent
    Public Overrides Sub DoSomething()
        MessageBox.Show("Hello from subclass")
        MyBase.DoSomething()
    End Sub
End Class
```

If we run our client code now, we’ll get two message boxes. First we’ll get the message from the subclass, followed by the one from the parent class.

The *MyBase* keyword can be used to invoke or use any *Public*, *Friend*, or *Protected* element from the parent class. This includes all of those elements directly on the base class, and also any elements the base class inherited from other classes higher in the inheritance chain.

*MyBase* only refers to the immediate parent of the current class. If we create a *SubSubClass* that inherits from *SubClass*, the *MyBase* keyword would refer to the *SubClass* code, not the *Parent* code. There is no direct way to navigate more than one level up the inheritance chain.

### 5.2.9 Overriding the Constructor Method

We’ve already seen how we can override methods, and how to use the *Me*, and *MyBase* keywords to interact with the various overridden methods in our inheritance chain. However, there are special rules that govern the process of overriding the *New* constructor method.

New methods aren’t automatically carried from a parent to a subclass like normal methods. Each subclass must define its own constructors, though those constructors may call into the parent class using the *MyBase* keyword:
When calling the base class constructor, that call must be the first line in our constructor code — anything else is an error. This is totally optional, however, since the constructor of the parent class is automatically called on our behalf before our constructor code begins to run, unless we make that call manually.

If all constructor methods of the base class require parameters then we must implement at least one constructor in our subclass and we must explicitly call MyBase.New from within our constructors.

As we discussed earlier, the New method can be overloaded, providing various implementations. If our parent class provides alternate implementations of New, we may want to manually make the call in order to cause the correct implementation to be called, based on our requirements.

5.3 Creating Base Classes and Abstract Methods

So far, we’ve seen how to inherit from a class, how to overload and override methods, and how virtual methods work. In all of our examples so far, the parent classes have been useful in their own right. Sometimes, however, we want to create a class such that it can only be used as a base class for inheritance.

5.3.1 MustInherit Keyword

Returning to our original sales order line item example, it may make little sense for anyone to create an object based on the generic LineItem class. In fact, we may want to ensure that only more specific subclasses derived from LineItem can be created. What we want to create is something called a base class. This is done using the MustInherit keyword in the class declaration:

Typically, no other Public MustInherit Class LineItem

change is required in our class. The result of this keyword is that it is no longer possible to write client code that creates an instance of the LineItem class, so the following would cause a syntax error:

    Dim obj As New LineItem()

Instead, to use the code in the LineItem class, we must create subclasses and use those throughout our application.

5.3.2 MustOverride Keyword

Another option we have is to create a method that must be overridden by a subclass. We might want to do this when we are creating a base class that provides some behavior, but relies on subclasses to also provide some behavior in order to function properly. This is accomplished by using the MustOverride keyword on a method declaration:

    Public MustOverride Sub CalcPrice()

Notice that there is no End Sub or any other code associated with the method.
When using `MustOverride`, we cannot provide any implementation for the method in our class. Such a method is called an abstract method or pure virtual function, since it only defines the interface and no implementation.

Methods declared in this manner must be overridden in any subclass that inherits from our base class. If we don’t override one of these methods, we’ll generate a syntax error in the subclass and it won’t compile.

### 5.3.3 Abstract Base Classes

We can combine these two concepts—using both `MustInherit` and `MustOverride`—to create something called an abstract base class. This is a class that provides no implementation, only the interface definitions from which a subclass can be created. An example might be as follows:

```vbnet
Public MustInherit Class Parent
    Public MustOverride Sub DoSomething()
    Public MustOverride Sub DoOtherStuff()
End Class
```

This technique can be very useful when creating frameworks or the high-level conceptual elements of a system. Any class that inherits `Parent` must implement both `DoSomething` and `DoOtherStuff` or a syntax error will result.

In some ways an abstract base class is very comparable to defining an interface using the `Interface` keyword. We’ll discuss the `Interface` keyword in detail later in this chapter. For now, be aware that the `Interface` keyword is used to formally declare an interface that can be implemented using the `Implements` keyword as in VB6.

We could define the same interface as shown in this example with the following code:

```vbnet
Public Interface IParent
    Sub DoSomething()
    Sub DoOtherStuff()
End Interface
```

Any class that implements the `IParent` interface must implement both `DoSomething` and `DoOtherStuff` or a syntax error will result—and in that regard this technique is similar to an abstract base class.

There are differences however. In particular, when we create a new class by subclassing the `Parent` class, that class can in turn be subclassed. Those classes will automatically have `DoSomething` and `DoOtherStuff` methods due to the nature of inheritance.

Compare this with the interface approach, where each individual class must independently implement the `IParent` interface and provide its own implementation of the two methods. If we never intend to reuse the code that implements these methods as we create new classes then the interface approach is fine, but if we want code reuse within subclasses inheritance is the way to go.

### 5.4 Shared or Class Members

While objects are very powerful and useful, there are times when we just want access to variables, functions, or routines that do useful work—without the need for an actual object instance. In the past, we would typically put this type of code into a simple code module even if the routine was technically related to some class.
5.4.1 Shared Methods

In VB.NET we have a better alternative. Not only can a class have all the regular methods and properties we’ve seen so far – methods and properties only available after creating an instance of the class – but they can also have methods that are available without creating an instance of the class. These are known as shared methods.

These methods are also known as static methods or class methods in other languages.

A shared method is not accessed via an object instance like a regular method, but rather is accessed directly from the class. The following is a simple example of a shared method:

```vbnet
Public Class Math
    Shared Function Add(ByVal a As Integer, ByVal b As Integer) As Integer
        Return a + b
    End Function
End Class
```

We can use this method – without instantiating a `Math` object – as follows:

```vbnet
Dim result As Integer
result = Math.Add(5, 10)
```

Notice how, rather than using an object variable, we use the actual class name to reference the method. With a normal method this would result in a syntax error, but with a shared method this is perfectly acceptable.

Shared methods can also be accessed via objects just like regular methods, but their most common use is to provide functionality without the requirement for creating an object. In fact, when a shared method is invoked, no object is created – the method is called directly, much like a procedure in a `Module`.

Shared methods can also be overloaded just like regular methods, so it is quite possible to create a set of variations on the same shared method, each having a different parameter list.

The default scope for a shared method is `Public`. It is possible to restrict the scope of a shared method to `Friend`, `Protected`, or `Private` by prefixing the declaration with the appropriate scope. In fact, when overloading a method we can have different scopes on each implementation – as long as the parameter lists are different as we discussed when covering the `Overloads` keyword earlier.

A good example of how shared methods are used comes from the .NET system class libraries. When we want to open a text file for input we typically make use of a shared method on the `File` class:

```vbnet
Dim infile As StreamReader = File.OpenText("words.txt")
Dim strIn As String
str = infile.ReadLine()
```

No object of type `File` is created here. The `OpenText` method is a shared method that opens a file and returns a `StreamReader` object for our use.
5.4.2 Shared Variables

There is another type of shared member we can create. There are times when it is nice to share a value across all instances of a class — when every object of a given type should share the same variable. This is accomplished through the use of shared variables.

A shared variable is declared using the *Shared* keyword, much like a shared method:

```
Public Class MyCounter
    Private Shared mintCount As Integer
End Class
```

As with shared methods, we can scope the shared variable as required. Where *Shared* methods are *Public* by default, *Shared* variables are *Private* by default.

*In general, it is good practice to always explicitly define the scope of methods and variables to avoid confusion.*

The important thing about shared variables is that they are common across all instances of the class. We could enhance our class slightly as follows:

```
Public Class MyCounter
    Private Shared mintCount As Integer
    Public Sub New()
        mintCount += 1
    End Sub
    Public ReadOnly Property Count() As Integer
        Get
            Return mintCount
        End Get
    End Property
End Class
```

As we create each instance of the class the counter is incremented by one.

At any point, we can retrieve the count value via the Count property. Thus, if we run the following client code we'll get a resulting value of 3:

```
Protected Sub Button4_Click(ByVal sender As Object, ByVal e As System.EventArgs)
    Dim obj As MyCounter
    obj = New MyCounter()
    obj = New MyCounter()
    obj = New MyCounter()
    MsgBox(obj.Count, MsgBoxStyle.Information, “Counter”)
End Sub
```

If we run it again we'll get 6, then 9, and so forth. As long as our application is running the counter will remain valid. Once our application terminates the counter also goes away.

This technique can be very useful for server processes that run “forever” since they can keep usage counters or other values over time very easily. The values are only reset when the process is restarted.
Another common use for shared variables is to provide a form of global variable. Given a Public scoped shared variable:

```visualbasic
Public Class TheClass
    Public Shared MyGlobal As Integer
End Class
```

We can then use this variable throughout our client code:

```
TheClass.MyGlobal += 5
```

This variable will be available to any code within our application, providing a very nice mechanism for sharing values between components, classes, modules, and so forth.
CHAPTER SIX

Exception Handling

Exceptions are runtime errors that occur when a program is running and causes the program to abort without execution. Such kind of situations can be handled using Exception Handling. By placing specific lines of code in the application we can handle most of the errors that we may encounter and we can enable the application to continue running.

VB .NET supports two ways to handle exceptions

- **Unstructured** - exception handling using the on error goto statement.
- **Structured** - exception handling using Try...Catch....Finally

Visual Basic .NET offers structured exception handling that provides a powerful, more readable alternative to "On Error Goto" error handling, which is available in previous versions of Microsoft Visual Basic. Structured exception handling is more powerful because it allows you to nest error handlers inside other error handlers within the same procedure. Furthermore, structured exception handling uses a block syntax similar to the If...Else...End If statement. This makes Visual Basic .NET code more readable and easier to maintain.

**NOTE**: Visual Basic .NET retains the "On Error Goto" syntax for backward compatibility. You can still use this syntax in Visual Basic .NET code. However, you cannot combine structured exception handling and On Error statements within the same procedure. When you write new code, Microsoft recommends that you use structured exception handling.

The basic syntax of structured error handling is as follows:

```vbnet
Try 'Code that may raise an error.
    Catch 'Code to handle the error.
    Finally 'Code to do any final clean up.
End Try
```

The **Try** and **End Try** statements are required. The **Catch** and **Finally** statements are not required, though you must include at least one of them in your code. You can also specify multiple **Catch** statements so that each **Catch** block handles a specific error.

Example One: **Catch an Exception**

```vbnet
Sub Main ()
    Dim a = 0, b = 1, c As Integer
    Try
        c = b / a
        'the above line throws an exception
        WriteLine("C is " & c)
    Catch e As Exception
        WriteLine(e)
        'catching the exception
    End Try
End Sub
```
Example Two: **Catch Multiple Exceptions**

```
Module Module1
    Sub Main()
        Dim a As Integer = 2147483647
        Dim b As Integer = 0
        Dim c As Integer = 0
        Try a += 1
            Catch exc As DivideByZeroException
                Console.WriteLine("Error: Divide by zero")
            Catch exc As OverflowException
                Console.WriteLine("Error: Overflow")
            Catch exc As Exception
                Console.WriteLine("Error: " & exc.Message)
        Finally Console.ReadLine()
    End Try
End Sub
End Module
```