Visual Basic Essentials

Student Guide

Revision 4.0

Object Innovations Course 4202

Visual Basic Essentials
Rev. 4.0

Student Guide

Information in this document is subject to change without notice. Companies, names and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Object Innovations.

Product and company names mentioned herein are the trademarks or registered trademarks of their respective owners.

® is a registered trademark of Object Innovations.

Authors: Robert J. Oberg and Dana Wyatt

Copyright ©2010 Object Innovations Enterprises, LLC All rights reserved..

Object Innovations
877-558-7246
www.objectinnovations.com

Printed in the United States of America.

Table of Contents (Overview)

Chapter 1	.NET: What You Need to Know
Chapter 2	Visual Basic for the Sophisticated Programmer
Chapter 3	Object-Oriented Programming in Visual Basic
Chapter 4	Visual Basic and the .NET Framework
Chapter 5	Delegates and Events
Chapter 6	Introduction to Windows Forms
Chapter 7	Newer Features in Visual Basic
Appendix A	Using Visual Studio 2010
Appendix B	Learning Resources

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Directory Structure

- **The course software installs to the root directory *C:\OIC\VbEss*.**
  - Example programs for each chapter are in named subdirectories of chapter directories **Chap01**, **Chap02**, and so on.
  - Example programs for Appendix A are in the directory **AppA**.
  - The **Labs** directory contains one subdirectory for each lab, named after the lab number. Starter code is frequently supplied, and answers are provided in the chapter directories.
  - The **CaseStudy** directory contains a case study in multiple steps.
  - The **Demos** directory is provided for performing in-class demonstrations led by the instructor.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Table of Contents (Detailed)

Chapter 1 .NET: What You Need to Know 1
Getting Started 3
.NET: What Is *Really* Happening 4
.NET Programming in a Nutshell 5
.NET Program Example 6
Viewing the Assembly 7
Viewing Intermediate Language 8
Understanding .NET 9
Creating a Console Application 10
Visual Studio Solutions 11
Starter Code 12
Using the Visual Studio Text Editor 13
IntelliSense 14
Build and Run the Project 15
Pausing the Output 16
Visual Basic and GUI Programs 17
.NET Documentation 18
Summary 19

Chapter 2 Visual Basic for the Sophisticated Programmer 21
Visual Basic 23
Hello, World 24
Compiling, Running (Command Line) 25
Program Structure 26
Namespaces 28
Project Imports 29
Startup Object 30
Variables 32
Literals and Constants 33
Operators 34
Short-Circuit Operators 35
More Operators 36
Operator Precedence 38
Control Structures 39
Conditionals 40
Looping Constructs 42
Types in Visual Basic 45
Object 46
Simple Data Types 47
Floating Point Data Types 48
Implicit Conversions 49
Explicit Conversions 50
Boolean Data Type 51

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Structure 52
Enumeration Types 53
Reference Types 54
Class Types 55
Object 56
String Data Type 57
Copying Strings 58
StringBuilder Class 59
Classes and Structures 61
Arrays 62
One Dimensional Arrays 63
System.Array 64
InputWrapper Class 65
Input Wrapper Implementation 66
Jagged Arrays 67
Rectangular Arrays 68
For Each for Arrays 69
Boxing and Unboxing 70
Output in Visual Basic 71
Formatting 72
Formatting Example 73
Modules 74
Subroutines and Functions 75
Default Parameters 78
Exceptions 79
Exceptions – Example Program 80
System.Exception 82
Implicit Line Continuation 83
Lab 2 84
Summary 85

Chapter 3 Object-Oriented Programming in Visual Basic 89
Visual Basic as an Object-Oriented Programming Language 91
Classes vs. Objects 92
Creating a Class 93
Creating and Referencing Objects 94
Bank Example 95
Account Class 96
Shared Members 98
Auto-Implemented Properties 100
Auto-Implemented Property Example 101
Classes and Modules 102
Self-Generating IDs 103
Methods vs. Properties 105
Defining Member Variables 106
Adding Methods 107
Adding Properties 108

 Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Overloading Methods 110
Object Lifecycle 112
Classes vs. Structures 116
Inheritance 118
Inheritance and Scope 120
Invoking the Base Class 122
Shadowing Base Class Methods 123
Polymorphism 124
Overriding Base Class Methods 125
Heterogeneous Collections 127
Abstract Classes 128
Abstract Methods 129
User Defined Exception Classes 130
Operator Overloading 133
Lab 3 135
Summary 136

Chapter 4 Visual Basic and the .NET Framework 139
Components and OO in Visual Basic 141
Interfaces 142
Interfaces in Visual Basic 144
Implementing an Interface 145
Using an Interface 146
Multiple Interfaces 147
Using Multiple Interfaces 150
TypeOf ... Is and Dynamic Interfaces 152
Interfaces in Visual Basic and COM 154
Resolving Ambiguity in Interfaces 155
System.Object 158
Collections 160
ArrayList 161
ArrayList Methods 162
Example: StringList 163
IEnumerable and IEnumerator 164
Using Enumerators 165
Collections of User-Defined Objects 166
Account Class 167
Collection Interfaces 168
ICollection 169
IList 170
Default Properties 171
Using the Item Property 172
Copy Semantics in Visual Basic 173
Arrays 174
Shallow Copy and Deep Copy 175
CopyDemo Example Program 176
Reference Copy 178

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Memberwise Clone 179
Using ICloneable 180
Lab 4A 181
Writing Generic Code 182
Using a Class of *Object* 183
Generic Types 184
Generic Example 185
Generic Client Code 186
System.Collections.Generic 187
Lab 4B 188
Summary 189

Chapter 5 Delegates and Events 195
Overview of Delegates and Events 197
Callbacks and Delegates 198
Usage of Delegates 199
Declaring a Delegate 200
Defining a Method 201
Creating a Delegate Object 202
Calling a Delegate 203
Random Number Generation 204
A Random Array 205
Combining Delegate Objects 206
Account.vb 207
DelegateAccount.vb 208
Events 209
Static and Dynamic Event Handling 210
Dynamic Event Handling 211
Chat Room Example 212
Static Event Handling 214
Lab 5 216
Summary 217

Chapter 6 Introduction to Windows Forms 221
Creating a Windows Forms App 223
Partial Classes 227
Windows Forms Event Handling 228
Add Events for a Control 229
Events Documentation 230
Closing a Form 231
ListBox Control 232
ListBox Example 233
My 234
Command Line Arguments 235
Lab 6 236
Summary 237

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Chapter 7 Newer Features in Visual Basic 241
Local Type Inference 243
Local Type Inference – Example 244
Object Initializers 245
Array Initializers 246
Anonymous Types 247
Partial Methods 248
Partial Method Definition 249
Partial Method Implementation 250
Test Program 251
Running the Example 252
Extension Methods 253
Extension Methods Example 254
Collection Initializers 255
Variance in Generic Interfaces 256
Covariance Example 257
Variance with IComparer(Of T) 258
Language-Integrated Query (LINQ) 259
LINQ Example 260
Using IEnumerable(OfT> 261
Summary 262

Appendix A Using Visual Studio 2010 263
A Visual Studio Solution 265
Toolbars 267
Customizing a Toolbar 268
Creating a Console Application 270
Using the Visual Studio Text Editor 271
Build and Run the Bytes Project 272
Running the Bytes Project 273
Executable File Location 274
Managing Configurations 275
Project Configurations 276
Debugging 277
Just-in-Time Debugging Demo 278
Breakpoints 281
Watch Variables 282
Debug Toolbar 283
Stepping with the Debugger 284
Demo: Stepping with the Debugger 285
Call Stack 286
Multiple-Project Solution Demo 287
Adding a Reference 288
Project Dependencies 289
Startup Project 290
Hidden Files 291
Summary 292

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Appendix B Learning Resources 293

 Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Chapter 1

.NET: What You Need to Know

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

.NET: What You Need to Know

Objectives

After completing this unit you will be able to:

- **Describe the essentials of creating and running a program in the .NET environment.**
- **Build and run a simple Visual Basic program in the .NET environment.**
- **Use the ILDASM tool to view intermediate language.**
- **Use Visual Studio 2010 as an effective environment for creating Visual Basic programs.**
- **Use the .NET Framework SDK documentation.**

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Getting Started

- **From a programmer's perspective, a beautiful thing about .NET is that you scarcely need to know anything about it to start writing programs for the .NET environment.**
  - You write a program in a high-level language (such as Visual Basic), a compiler creates an executable .EXE file (called an **assembly**), and you run that .EXE file.
- **Even very simple programs, if they are designed to do something interesting, such as perform output, will require that the program employ the services of library code.**
  - A large library, called the .NET Framework Class Library, comes with .NET, and you can use all of the services of this library in your programs.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

.NET: What Is *Really* Happening

- **The assembly that is created does not contain executable code, but, rather, code in Intermediate Language, or IL (sometimes called Microsoft Intermediate Language, or MSIL).**
  - In the Windows environment, this IL code is packaged up in a standard portable executable (PE) file format, so you will see the familiar .EXE extension (or, if you are building a component, the .DLL extension).
  - You can view an assembly using the **ILDASM** tool.
- **When you run the .EXE, a special runtime environment (the Common Language Runtime, or CLR) is launched and the IL instructions are executed by the CLR.**
  - Unlike some runtimes, where the IL would be interpreted each time it is executed, the CLR comes with a just-in-time (JIT) compiler, which translates the IL to native machine code the first time it is encountered.
  - On subsequent calls, the code segment runs as native code.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

.NET Programming in a Nutshell

1. Write your program in a high-level .NET language, such as Visual Basic.
2. Compile your program into IL.
3. Run your IL program, which will launch the CLR to execute your IL, using its JIT to translate your program to native code as it executes.

- **We will look at a simple example of a Visual Basic program, and run it under .NET.**
  - Don’t worry about the syntax of Visual Basic, which we will start discussing in the next chapter.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

.NET Program Example

- **See *SimpleCalc* in the *Chap01* folder.**

1. Enter the program in a text editor.

```
' SimpleCalc.vb
'
' This program does a simple calculation: calculate
' area of a rectangle

Module SimpleCalc

   Sub Main ()
      Dim base As Integer = 20
      Dim height As Integer = 5
      Dim area As Integer
      area = base * height
      System.Console.WriteLine("Area = {0}", area)
   End Sub

End Module
```

2. Compile the program at the command line. Start the console window via Start | All Programs | Microsoft Visual Studio 2010 | Visual Studio Tools | Visual Studio Command Prompt (2010). Navigate to the **Chap01\SimpleCalc** folder.

```
>vbc SimpleCalc.vb
```

3. Run your IL program **SimpleCalc.exe**

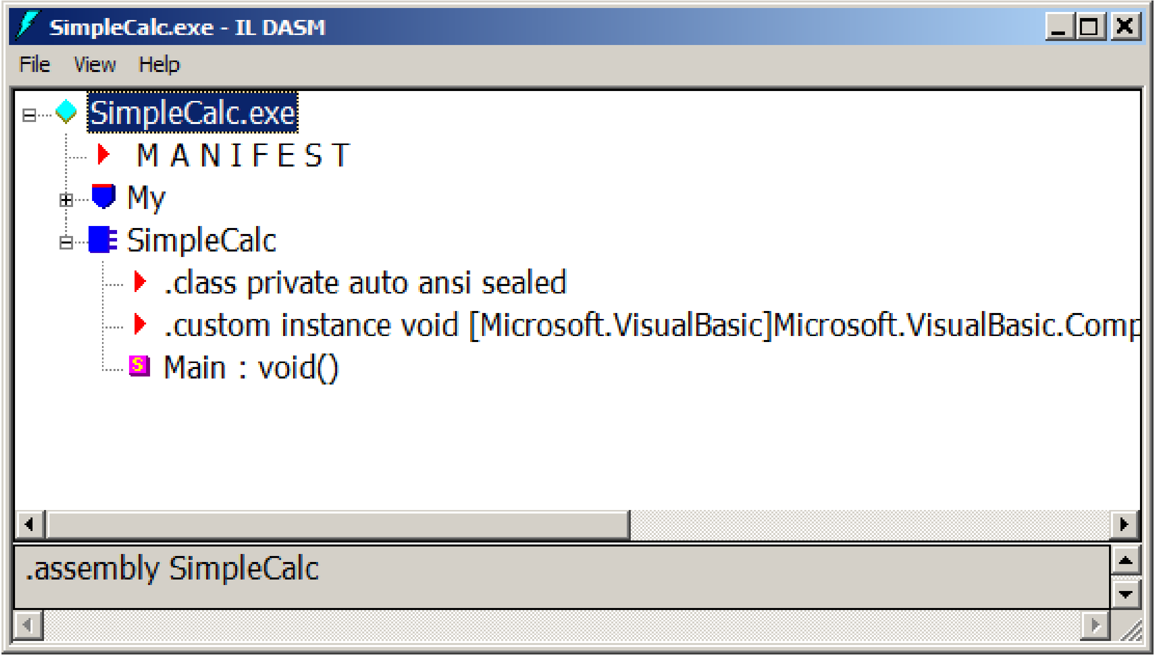
```
>SimpleCalc
area = 100
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Viewing the Assembly

- **You can view the assembly using the *ILDASM* tool.**

```
>ildasm SimpleCalc.exe
```



Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Viewing Intermediate Language

- **Double-click on *Main:void()***

SimpleCalc::Main : void()

Find Find Next

```
.method public static void  Main() cil managed
{
  .entrypoint
  .custom instance void [mscorlib]System.STAThreadAttribute::.ctor() = (
  // Code size       26 (0x1a)
  .maxstack  2
  .locals init (int32 V_0,
           int32 V_1,
           int32 V_2)
  IL_0000:  ldc.i4.s   20
  IL_0002:  stloc.1
  IL_0003:  ldc.i4.5
  IL_0004:  stloc.2
  IL_0005:  ldloc.1
  IL_0006:  ldloc.2
  IL_0007:  mul.ovf
  IL_0008:  stloc.0
  IL_0009:  ldstr      "Area = {0}"
  IL_000e:  ldloc.0
  IL_000f:  box        [mscorlib]System.Int32
  IL_0014:  call       void [mscorlib]System.Console::WriteLine(string,
                                                                 object)
  IL_0019:  ret
} // end of method SimpleCalc::Main
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Understanding .NET

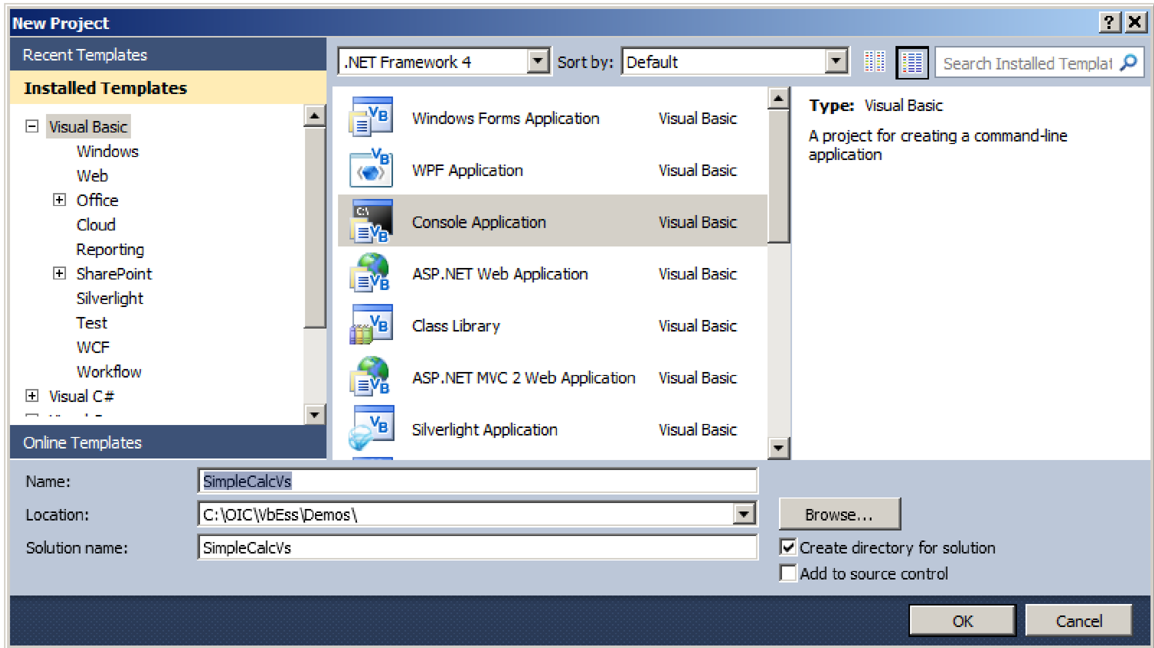
- **The nice thing about a high-level programming language is that you usually do not need to be concerned with the platform on which the program executes.**
- **You can work with the abstractions provided by the language and with functions provided by libraries.**
- **Your appreciation of the Visual Basic programming language and its potential for creating great applications will be richer if you have a general understanding of .NET.**
- **After this course, we suggest you next study:**
  - .NET Framework Using Visual Basic
- **After that there are many courses, such as:**
  - Windows Presentation Foundation Using Visual Basic
  - ASP.NET Using Visual Basic
  - Silverlight 4 Using Visual Basic

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Creating a Console Application

- **We will now create a simple console application.**
  - Our program is the same simple calculator we created earlier that was compiled at the command line.

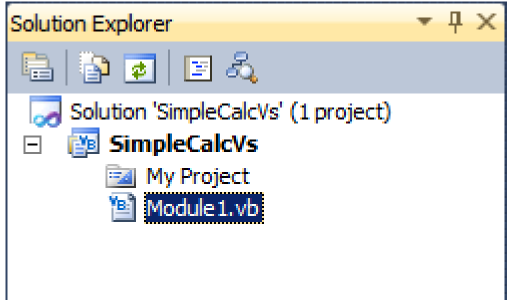
1. From the Visual Studio main menu, choose File | New Project.... This will bring up the New Project dialog.
2. Choose "Console Application."
3. In the Name field, type **SimpleCalcVs** and for Location browse to **C:\OIC\VbEss\Demos**. Click OK.



Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Visual Studio Solutions

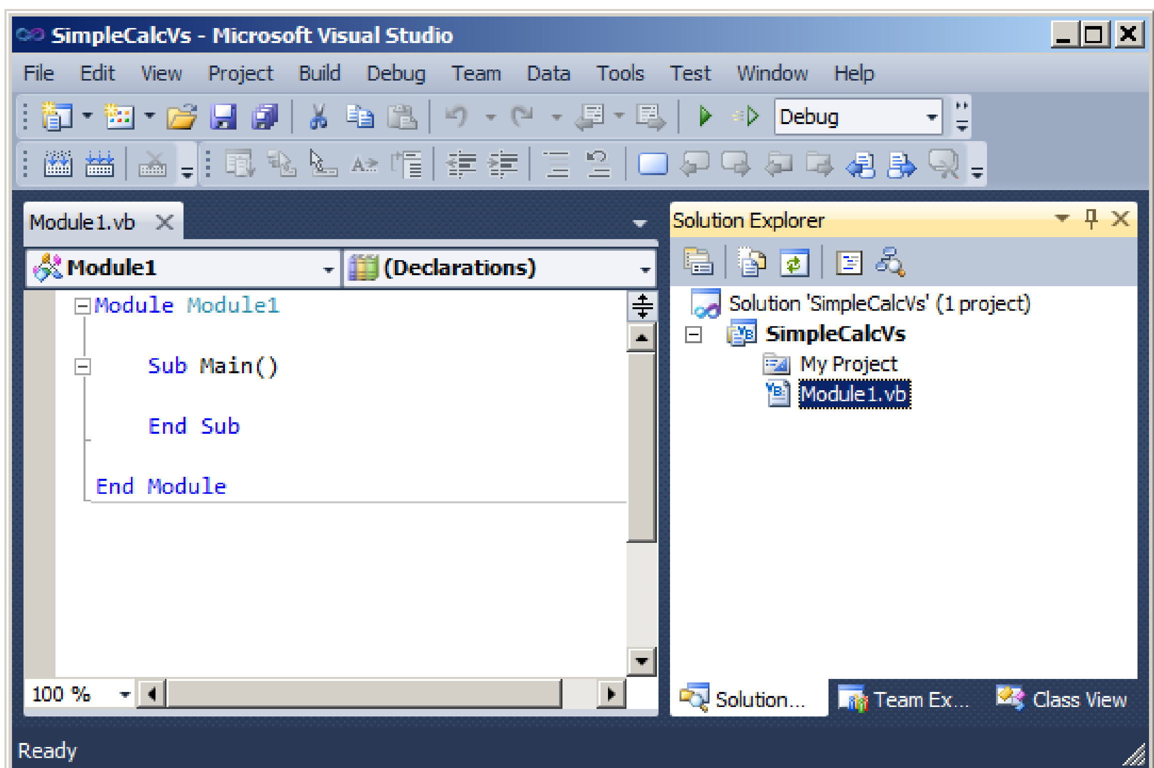
- **In Visual Studio 2010, project information is organized by *solutions* and *projects*.**
  - A solution, specified by a **.sln** file, consists of one or more projects, specified by **.vbproj** files in the case of Visual Basic.
- **Notice Solution Explorer in the top-right.**



 Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Starter Code

- **We see that a Visual Studio solution has been created with one project.**
- **The project contains two elements.**
  - The folder **MyProject** contains a number of files that we normally will not need to touch.
  - **Module1.vb** contains skeleton code that we will edit.



  - We've closed a few windows that we don't need at this point.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Using the Visual Studio Text Editor

- **In Solution Explorer, change the name of the file *Module1.vb* to *SimpleCalc.vb*.**
- **Other changes will be made for you automatically, such as changing the name of the module to *SimpleCalc*.**
- **Make the following edits, using the Visual Studio text editor.**

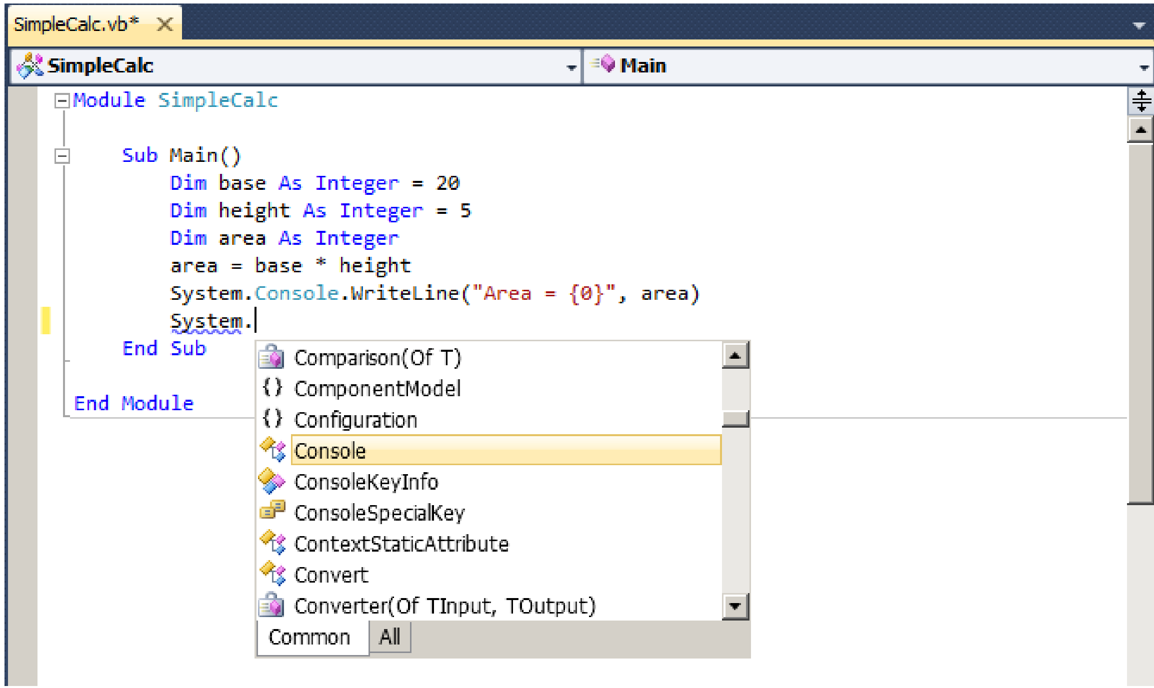
```
Module SimpleCalc

   Sub Main()
      Dim base As Integer = 20
      Dim height As Integer = 5
      Dim area As Integer
      area = base * height
      System.Console.WriteLine("Area = {0}", area)
   End Sub

End Module
```

  - Notice that the Visual Studio text editor highlights syntax, indents automatically, and so on.
  - In Visual Basic, as opposed to Visual C#, the editor does other things for you too, such as supply matching End keywords, adjust capitalization of keywords, and so on.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

IntelliSense

- **A powerful feature of Visual Studio is *IntelliSense*.**
  - IntelliSense will automatically pop up a list box allowing you to easily insert language elements directly into your code.



 Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Build and Run the Project

- **Building a project means compiling the individual source files and linking them together with any library files to create an IL executable .EXE file.**
- **You can build the project by using one of the following:**
  - Menu Build | Build Solution or toolbar button or keyboard shortcut Ctrl+Shift+B.
  - Menu Build | Build SimpleCalcVs or toolbar button (this just builds the project SimpleCalcVs)[1].
- **You can run the program without the debugger by using one of the following:**
  - Toolbar (This toolbar button is not provided by default; see Appendix A for how to add it to your Build toolbar.)
  - Keyboard shortcut Ctrl + F5
- **You can run the program in the debugger by using one of the following:**
  - Menu Debug | Start Debugging
  - Toolbar
  - Keyboard shortcut F5
- **Try it!**

[1] The two are the same in this case, because the solution has only one project, but some solutions have multiple projects, and then there is a difference.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Pausing the Output

- **If you run the program in the debugger from Visual Studio, you will notice that the output window automatically closes on program termination.**
- **To keep the window open, you may prompt the user for some input.**

```
Module SimpleCalc

   Sub Main()
      Dim base As Integer = 20
      Dim height As Integer = 5
      Dim area As Integer
      area = base * height
      System.Console.WriteLine("Area = {0}", area)
      System.Console.WriteLine( _
       "Prese Enter to exit")
      System.Console.ReadLine()
   End Sub

End Module
```

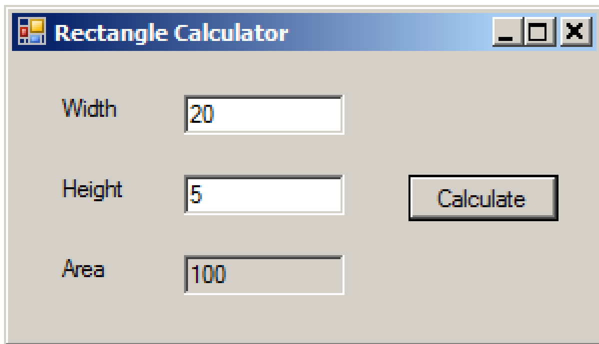
- **This version of the program is saved as a Visual Studio 2010 project in *Chap01\SimpleCalcVs*.**
- **Remember that you can always make the console window stay open by running without the debugger via Control + F5.**

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Visual Basic and GUI Programs

- **Microsoft's Visual Basic language works very effectively in a GUI environment.**
  - Using Windows Forms, it is easy to create Windows GUI programs in Visual Basic.

Example: See **Chap01\SimpleCalcGui**

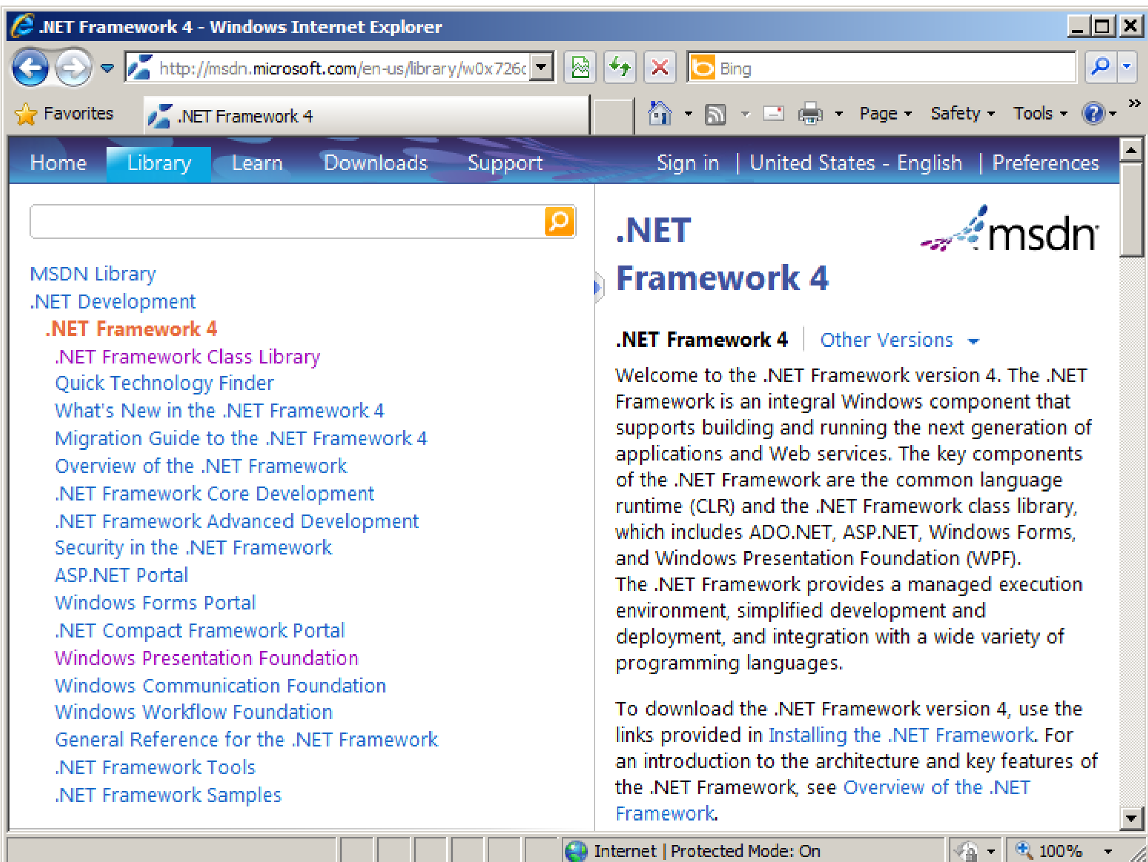


- **We will discuss GUI programming using Visual Basic in Chapter 6.**

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

.NET Documentation

- **MSDN documentation for the .NET Framework is included with Visual Studio 2010.**
  - Use the menu Help | View Help.



- **You may also find the documentation on the Web:**

http://msdn.microsoft.com

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Summary

- **As in other environments, with .NET you write a program in a high-level language, compile to an executable (.EXE file), and run that .EXE file.**
- **The .EXE file, called an *assembly*, contains Intermediate Language instructions.**
- **You can view an assembly through the *ILDASM* tool.**
- **Visual Studio 2010 is a powerful IDE that makes it easy to develop Visual Basic programs.**
- **With Visual Studio, it is easy to create GUI programs using Visual Basic.**
- **You can access extensive .NET Framework documentation through the Visual Studio help system.**

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

 Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Chapter 2

Visual Basic for the Sophisticated Programmer

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Visual Basic for the Sophisticated Programmer

Objectives

After completing this unit you will be able to:

- **Compile and run Visual Basic programs in your local development environment.**
- **Describe the basic structure of Visual Basic programs.**
- **Describe how related Visual Basic classes can be grouped into namespaces.**
- **Describe objects and classes in Visual Basic.**
- **Perform input and output in Visual Basic.**
- **Outline the principle control structures and operators in Visual Basic.**
- **Outline the principle data types in Visual Basic.**
- **Describe the difference between value and reference types, and explain how Visual Basic achieves a unified type system through "boxing" and "unboxing."**
- **Use structures, strings and arrays.**
- **Perform formatting in Visual Basic.**
- **Use exceptions in Visual Basic.**

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Visual Basic

- **Visual Basic is a powerful, flexible programming language.**
  - It supports a variety of data types, operators, and control structures.
  - It can be used to create user-defined classes, and it supports inheritance.
  - It supports structured exception handling.
  - It can be used to create threads.
- **To take full advantage of the .NET Framework, Visual Basic makes some significant changes to the VB6 programming model.**
  - This will present some challenges to VB6 programmers, but migration tools should assist in the process.
- **As a terminology note, beginning with .NET 2.0, Microsoft has dropped the ".NET" in the Visual Basic language.**
  - The pre-.NET version of the language is now referred to as Visual Basic 6 or VB6.
  - In this course, Visual Basic, VB, Visual Basic 2010, Visual Basic .NET, and VB.NET are all synonymous.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Hello, World

- **Whenever learning a new programming language, a good first step is to write and run a simple program that will display a single line of text.**
  - Such a program demonstrates the basic structure of the language, including output.
  - You must learn the pragmatics of compiling and running the program.
- **Here is "Hello, World" in Visual Basic:**
  - See **Hello\Hello.vb**.

```
' Hello.vb

Module Hello

   Sub Main()
      System.Console.WriteLine( _
         "Hello, world")
   End Sub

End Module
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Compiling, Running (Command Line)

- **The Visual Studio 2010 IDE (integrated development environment) was introduced in Chapter 1, and we will use it throughout the course.**
  - See Appendix A for more details.
- **If you are using the .NET SDK, you may do the following:**
  - Compile the program via the command line:

```
vbc Hello.vb
```

  - An executable file **Hello.exe** will be generated. To execute your program, type at the command line:

```
Hello
```

  - The program will now execute, and you should see the greeting displayed. That's all there is to it!

```
Hello, World
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Program Structure

```
' Hello.vb

Module Hello

   Sub Main()
      System.Console.WriteLine( _
         "Hello, world")
   End Sub

End Module
```

- **The program begins with a *comment*.**
  - A single quote mark is used to indicate the beginning of a comment; the remainder of the line is ignored by the compiler.
- **Console applications contain a *module* that has a *Sub Main*.**
  - **Sub Main** is the *Startup object* and is called when the program begins.
  - In Visual Basic such a unit of code is called a *procedure* or a *method*.
- **Program units such as *Module* and *Sub* have matching *End*.**
  - **End Module** and **End Sub**.
- **Visual Basic files have the extension *.vb*.**

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Statements

```
' Hello.vb

Module Hello

   Sub Main()
      System.Console.WriteLine( _
          "Hello, world")
   End Sub

End Module
```

- **Every method in Visual Basic has zero or more *statements*.**
- **A statement is terminated by a new line**
  - A statement may be continued onto the following line by using one or more spaces followed by the underscore character.
  - In Visual Basic 2010 a new feature of *implicit line continuation* (discussed later) allows you to continue a statement on the next line without the underscore character.
- **The *Console* class provides support for standard output and standard input.**
  - The method **WriteLine()** displays a string, followed by a new line.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Namespaces

- **Much standard functionality in Visual Basic is provided through many *classes* in the .NET Framework.**
- **Related classes are grouped into *namespaces*.**
- **The fully qualified name of a class is specified by the namespace, followed by a dot, followed by class name.**

```
System.Console
```

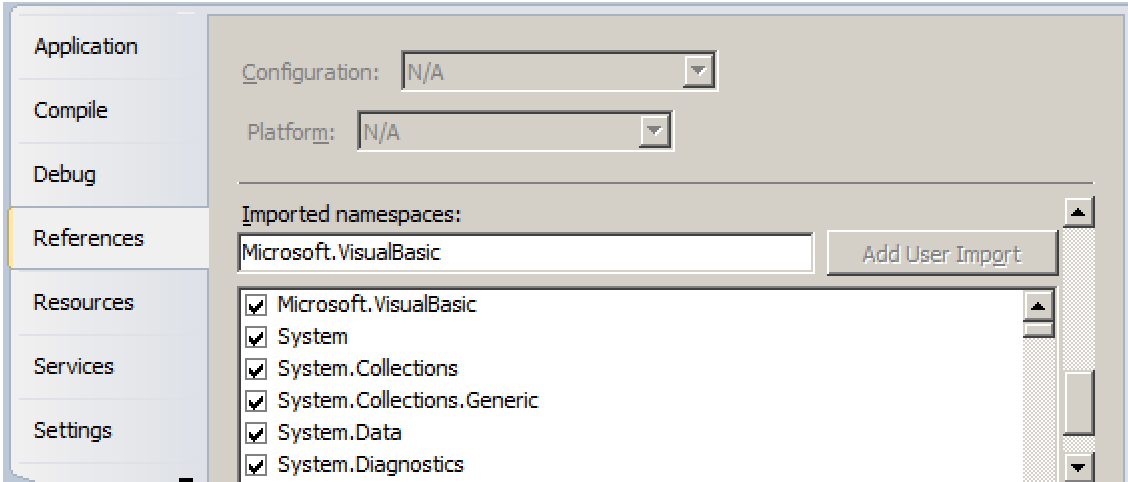
- **An *Imports* statement allows a class to be referred to by its class name alone.**
  - See **Hello2\Hello2.vb**.

```
' Hello2.vb

Imports System

Module Hello

   Sub Main()
      Console.WriteLine("Hello, world")
   End Sub

End Module
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Project Imports

- **Namespaces can be imported implicitly for all files in the project by listing them in the Project Imports.**
  - Right-click on the project and choose Properties. Select References from the list on the left, and scroll down until you find Imported namespaces.



- **See *Hello3*, where we omit explicitly importing the System namespace, but it is still found.**

```
' Hello3.vb

Module Hello

   Sub Main()
      Console.WriteLine("Hello, world")
   End Sub

End Module
```

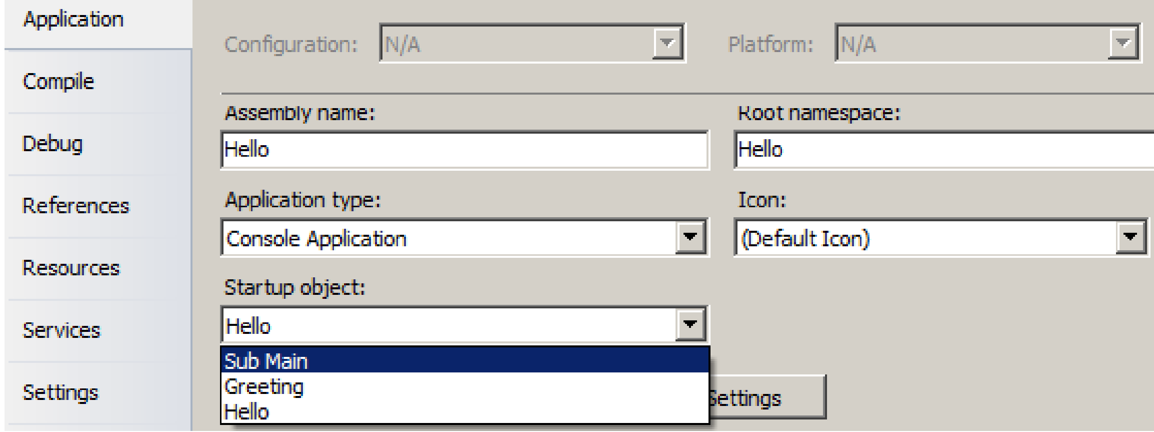

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Startup Object

- **Another important project setting is the Startup object.**
- **To see the importance of it, open up the *Hello* project in the *Demos* folder.**
  - This is a copy of the original **Hello** program. Build and run it.
  - Now, change the name of the module from **Hello** to **Greeting**. Build it. You'll get an error!

```
'Sub Main' was not found in 'Hello.Hello'.
```

- **To fix this problem, right-click on *Hello* in Solution Explorer and select Properties from the context menu.**



  - Using the dropdown list, make **Sub Main** the startup object.
  - Build and run. Now it should work!
  - This alternate version of our program is saved in the **Greeting** directory of this chapter.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Microsoft.VisualBasic Namespace

- **Microsoft provides functionality for many VB6 features through the *Microsoft.VisualBasic* namespace.**
  - The **Constants** module provides many common **VB6** constants (for example, **vbFalse**, **vbOkCancel**, **vbWednesday**, and so on.)
  - The **ControlChars** module provides many constants (for example, **Tab**, **Quote**, **CrLf**, and so on.)
  - The **DateAndTime** module has many common **VB6** date/time properties and functions (for example, **Now**, **Today**, **DateSerial()**, and so on.)

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Variables

- **In Visual Basic, you can define *variables* to hold data.**
- **Variables represent storage locations in memory.**
- **In Visual Basic, variables are of a specific data *type*.**
  - Some common types are **Integer** for integers and **Double** for floating point numbers.
  - You should declare variables before using them.
- **A variable declaration reserves memory space for the variable and may optionally specify an initial value.**

```
Dim fahr As Integer = 86  ' reserves space and
                          ' assigns an intial value
Dim celsius As Integer    ' reserves space but does
                          ' not initialize
Dim first, last As String ' both are strings
```

  - If an initial value is not specified, Visual Basic initializes the variable to a default value, such as 0.
- **Visual Basic allows you to select whether variable declarations are required.**
  - You may set **Option Explicit On** (declarations required) or **Off** (declarations not required) by setting the project's Build properties.
  - If **Option Explicit** is off, the compiler assumes that all undeclared variables are of type **Object.**

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Literals and Constants

- **In Visual Basic, literals are typed.**

```
12              ' Integer literal
9.2             ' Double literal
"Hi"            ' String literal
False           ' Boolean literal
```

  – To avoid ambiguity, a suffix can be placed on the literal to specify its type.

```
12S             ' Short literal
12I             ' Integer literal
12L             ' Long literal
12F             ' Single (float) literal
12R             ' Double literal
12D             ' Decimal literal
"H"c            ' Char literal
```

- **In Visual Basic, constants can also be declared.**

```
Private Const ONE As Integer = 1
Public Const PI As Double = 3.14159
```

  – The **Public** and **Private** keywords determine the scope of the constant and are discussed in another section.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Operators

- **Visual Basic .NET has a richer set of operators than any previous version of VB.**
- **There are arithmetic operators:**

```
Dim a, b, c  As Single
Dim num As Integer
a = b * c                       ' multiplication
a = b / c                       ' division
num = b.ToInt16() \ 3           ' integer division
num = b.ToInt32() Mod 3         ' modulo (remainder)
a = CSng (b ^ 2)                ' exponentiation
a = b + c                       ' addition
a = b - c                       ' subtraction
a = -b                          ' negation
```

- **There are relational operators:**

```
Dim a, b, c  As Single
If a = b Then ...     ' equality
If a <> b Then ...    ' inequality
If a < b Then ...     ' less than
If a <= b Then ...    ' less than or equal to
If a >= b Then ...    ' greater than or equal to
If a > b Then ...     ' greater than

If a <> 0 And b/a < 10 Then ...      ' And
If A < 0 Or b > 100 Then ...         ' Or
If Not a = b Then ...                ' Not
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Short-Circuit Operators

- **The *AndAlso* operator, like the *And* operator, does logical conjunction, but it is *short-circuited*.**
  - If the first expression evaluates to **False**, the rest of the expression is not evaluated, because it cannot affect the final result, and **AndAlso** will return **False**.
- **Similarly, the *OrElse* operator, which does logical disjunction like the *Or* operator, is short-circuited.**
  - If the first expression evaluates to **True**, the rest of the expression is not evaluated, because it cannot affect the final result, and **OrElse** will return **True**.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

More Operators

- **There is also a *TypeOf...Is* operator:**

```
Sub TestEmployee (obj As Object)

  If TypeOf obj Is Employee Then
     ' Do something
  ElseIf TypeOf obj Is String Then
     ' Do something
  End If

End Sub
```

- **Visual Basic has bitwise operators:**

```
Dim a, b, c  As Byte
b = 15
c = 255
...
a = b And c       ' bitwise And  (a = 15)
a = b Or c        ' bitwise Or   (a = 255)
a = b Xor c       ' bitwise Xor  (a = 240)
a = Not b         ' ones complement  (a = 240)
```

- **There are shift operators:**

```
result = pattern << amount ' arithmetic left shift

result = pattern >> amount ' arithmetic right shift
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

More Operators (Cont'd)

- **There are assignment operators:**

```
Dim a, b  As Single
Dim num As Integer
a = b                        ' assignment
a *= b + 4                   ' multiply/assign
a /= b                       ' divide/assign
num \= 3                     ' integer divide/assign
a += b * 2                   ' add/assign
a -= b                       ' subtract/assign
```

- **There are string operators:**

```
Dim a, b  As String
a = "Red "                   ' assignment
b = a & "Light. "            ' concatenation
b &= " Stop!"                ' concatenate/assign
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Operator Precedence

- **A precedence table is shown below, from highest to lowest.**
  - Within a level, operators are evaluated from left to right.

Category	Operators
Arithmetic	^
	- (negative) + (positive)
	* /
	\
	Mod
	+ -
	& (concatenation)
	<< >>
Comparison	== <> < <= > >= Is IsNot Like TypeOf...Is
Conditional	Not
	And AndAlso
	Or OrElse
	Xor

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Control Structures

- **Visual Basic has a powerful set of control structures and functions that include:**

CONDITIONALS

- **If** statement
- **IIf** function
- **Select** statement

LOOPING

- **While** statement
- **Do** statement
- **For** statement

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Conditionals

- **The *If* statement is similar to that found in most languages:**

```
' Single-line if

  If a < 0 Then a = 0
  If a < 0 Then resp = "No" Else resp = "Yes"

' If ... End If

  If a < 0 Then
    resp = "No"
    a = 0
  End If

  If a < 0 Then
    resp = "No"
    a = 0
  Else
    resp = "Yes"
  End If

' If .. ElseIf .. End If

  If a < 0 Then
    resp = "No"
    a = 0
  ElseIf a = 0 Then
    resp = "Maybe"
  Else
    resp = "Yes"
  End If
```

 Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Conditionals (Cont'd)

- **The *IIf* (immediate *If*) function can be used to select a value to be used within an expression:**

```
resp = IIf (a < 0, "No", "Yes")
```

  - If the condition is true, the second parameter is returned; if the condition is false, the third parameter is returned.

- **The *Select* statement is used to execute a set of statements, depending on the value of an expression.**
  - The expression can be any numeric or string expression.

```
Select Case testScore
Case 100                ' single value
  grade = "A+"
Case Is >= 92           ' Is <operator> value
  grade = "A"
Case 90, 91             ' list of values
  grade = "A-"
Case 88 To 89           ' range of values
  grade = "B+"
Case Else               ' default
  grade = "Who cares"
End Select
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Looping Constructs

- **Visual Basic has several versions of a *While* statement:**

```
' While ... End While

  While totWeight < limit
    totWeight += item(i).weight
    i = i + 1
  End While
```

  - NOTICE: VB6 programmers should note that the end of the **While** statement is now **End While** instead of **Wend.**

```
' Do While ... Loop

  Do While totWeight < limit
    totWeight += item(i).weight
    i = i + 1
  Loop


' Do Until ... Loop   (the opposite of a While)

  Do Until limit >= totWeight
    totWeight += item(i).weight
    i = i + 1
  Loop
```

  - The advantage of using the **Do** control structure is that it has an **Exit Do** statement.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Looping Constructs (Cont'd)

```
' Do While ... Loop   with Exit Do

  Dim buf As String
  Dim lineNum As Integer = 1

  Do While lineNum < 5
    buf = Console.ReadLine ()
    If buf.StartsWith("EXIT") Then Exit Do
    Console.WriteLine("Line " + _
      lineNum.ToString() + ": " + buf)
    lineNum += 1
  Loop
```

- **Visual Basic also has a version of the *Do* statement that tests the condition at the bottom of the loop:**

```
' Do ... Loop While

  Do
    ' loop statements
  Loop While lineNum < 5


' Do ... Loop Until

  Do
    ' loop statements
  Loop Until lineNum >= 5
```

- **Visual Basic has a *Continue* statement, which will immediately skip to the next iteration of a loop.**
  - See example program **LoopContinue**.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Looping Constructs (Cont'd)

- **Visual Basic also has a *For* statement:**

```
For i = 1 To count
   ' loop statements
Next


For i = count To 1 Step -1
   ' loop statements
Next


' The For has an Exit For that can be used
' for early exit from the loop
  For i = 1 to 100
     ' loop statements
     If someCondition Then Exit For
     ' more loop statements
  Next


' If nesting For's, use the loop index in the Next
' for readability
  For i = 1 to 100
     For j = 1 to 5
        ' loop statements
     Next j
  Next i
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Types in Visual Basic

- **In Visual Basic there are two kinds of types:**
  - Value types
  - Reference types
- **Value types directly contain their data.**
  - Each variable of a value type has its own copy of the data.
  - Value types are typically allocated on the stack and get automatically destroyed when the variable goes out of scope.
- **Reference types do not contain data directly but only “refer” to data.**
  - Variables of reference types store *references* to data, called *objects*.
  - Two different variables can reference the same object.
  - Reference types are typically allocated on the heap, and eventually get destroyed through a process known as *garbage collection*[1].

[1] For a discussion of garbage collection, including a programming example, see Chapter 6 of Object Innovations’ course 4212, .NET Framework Using Visual Basic

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Object

- **All types in Visual Basic are derived from *Object*.**
  - This means that any native data types can be substituted whenever a subroutine or function expects an **Object**.
  - An **Object** variable occupies 4 bytes of memory and contains a reference to the actual variable.
  - The **Object** class contains the method **GetType** to inquire about the type of variable that the object variable references.
  - NOTICE: VB6 programmers should note that the **Variant** type is no longer supported.

```
Sub TestGetType (obj As Object)

  Dim st as System.Type
  st = obj.GetType()
  Select Case st.ToString()
  case "System.Int32"
    Console.WriteLine("Number is: " & _
      obj.ToString() )
  case "System.String"
    Console.WriteLine("String is: " & _
      obj.ToString() )
  End Select

End Sub
```

  - Calls to **TestGetType** would resemble:

```
TestGetType (42)
TestGetType ("Forty two")
```

  - See the example program **DemoGetType**.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Simple Data Types

- **The simple data types include:**

Visual Basic TYPE	.NET FRAMEWORK TYPE	MEMORY REQUIREMENTS	RANGE
Object	**System.Object**	4 bytes	Any type can be stored in a variable of type **Object**
Boolean	**System.Boolean**	4 bytes	**True** or **False**
Byte	**System.Byte**	1 byte	0 to 255 (unsigned)
Char	**System.Char**	2 bytes	0 to 65,535 (unsigned)
Short	**System.Int16**	2 bytes	-32,768 to 32,767
Integer	**System.Int32**	4 bytes	-2,147,483,648 to 2,147,483,647
Long	**System.Int64**	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
Single	**System.Single**	4 bytes	single-precision floating point corresponding to the IEEE 754 standard
Double	**System.Double**	8 bytes	double-precision floating point corresponding to the IEEE 754 standard
String	**System.String**	10 bytes + (2 * string length)	0 to almost 2.1 billion Unicode characters
Decimal	**System.Decimal**	12 bytes	96-bit signed integers scaled by a variable power of 10; from 0 to 28 digits to the right of the decimal point
Date	**System.DateTime**	8 bytes	January 1, 1 CE (year 1) to December 31, 9999 and times from 00:00:00 to 23:59:59

- **Notice to VB6 Programmers:**
  - The fixed-length string (**String*20**) from earlier versions of VB is no longer supported.
  - **Currency** is no longer supported; it has been replaced by **Decimal**.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Floating Point Data Types

- **Visual Basic supports the following predefined floating point data types.**
  - The **Single** data type is a single-precision floating point.
  - The **Double** data type is a double-precision floating point.
- **The *Single* data type is represented in the IEEE 754 32-bit single-precision floating point format.**
- **The *Double* data type is represented in the IEEE 754 64-bit double-precision floating point format.**
- **IEEE 754 define the following special floating point values:**
  - **Positive zero** results from dividing 0.0 by a non-zero positive value.
  - **Negative zero** results from dividing 0.0 by a non-zero negative value.
  - **Positive infinity** results from dividing a non-zero positive value by 0.0.
  - **Negative infinity** results from dividing a non-zero negative value by 0.0.
  - **Not-a-Number (**also known as **NaN**) results from dividing 0.0 by 0.0.

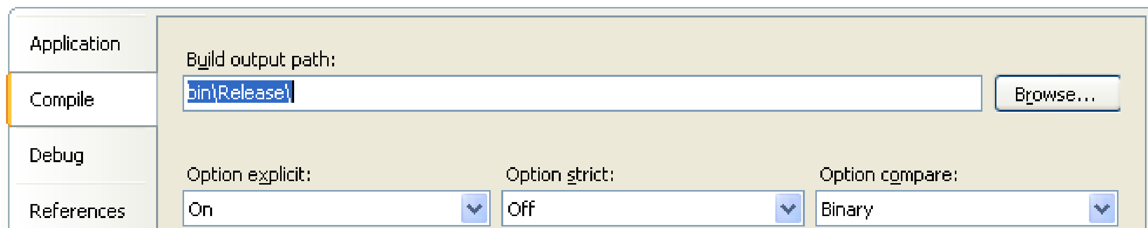
Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Implicit Conversions

- **Visual Basic can be configured to allow or disallow most implicit conversions.**
  - An implicit conversion is one in which VB must convert a value used in an expression to another type in order for the operation to be performed.
  - A conversion is safe if there is no loss of precision (for example, conversions from **Integer** to **Long**).
  - A conversion might cause problems if there is a loss of precision (for example, conversions from **Single** to **Long**).

```
Dim num As Integer
Dim val As Single

num = 12.3          ' unsafe
val = 8             ' safe
```

  - You may set **Option Strict On** (non-safe implicit conversions not allowed) or **Off** (implicit conversions allowed) by setting the project's Compile properties.



Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Explicit Conversions

- ***Explicit conversions* are performed only where the programmer uses a cast expression explicitly.**
- **Explicit conversions are risky in that loss of information can easily occur.**
- **Methods for explicit conversions include:**
  - The native Visual Basic cast functions such as **CInt**, **CLng**, **CSng**, **CDbl**, **CStr**, etc.
  - The new .NET Framework class **Convert**.
  - For an example, see the program **DemoConvert**.

```
' DemoConvert.vb

Module DemoConvert

   Sub Main()
      Dim buf As String = "12.3"
      Dim num1 As Single = CSng(buf)
      Dim num2 As Integer = CInt(num1)
      Dim num3 As Byte = Convert.ToByte(num1)
      Console.WriteLine("string = {0}", buf)
      Console.WriteLine("single = {0}", num1)
      Console.WriteLine("integer = {0}", num2)
      Console.WriteLine("byte = {0}", num3)
   End Sub

End Module
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Boolean Data Type

- **The *Boolean* data type represents a true/false value.**
  - Boolean values are also known as logical values, and may only be set to the values **True** or **False**.
- **No predefined conversions exist between *Boolean* and other types.**
  - In Visual Basic you have to explicitly use relational operators.

```
If numTemp = 0 Then
     ...

If str <> "hello" Then
     ...
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Structure

- **A *Structure* is a value type which can group inhomogeneous types together.**
  - It can also have constructors and methods, which we will look at later.

```
Public Structure Hotel
   Public City As String
   Public Name As String
   Public Rooms As Integer
   Public Cost As Decimal
End Structure
```

- **A structure object is created using the *New* operator.**

```
Dim h As New Hotel()
```

- **A structure object can also be created without *New*, but then the fields will be unassigned, and the object cannot be used until the fields have been initialized.**

```
Dim h As Hotel
h.Name = "Sheraton"
'  Now it is OK to use h.Name field
h.City = "Atlanta";
h.Rooms = 100
h.Cost = 50.00m;
'  Now it is OK to use h object
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Enumeration Types

- **The final kind of value type is an *enumeration* type.**
- **An enumeration type is a distinct type with named constants.**
- **Every enumeration type has an underlying type, which is one of *Byte*, *Short*, *Integer*, *Long*.**
- **An enumeration type is defined through an *Enum* declaration.**

```
Public Enum BookingStatus As Byte
    HotelNotFound,         ' 0 implicitly
    RoomsNotAvailable,     ' 1 implicitly
    Ok = 5                 ' explicit value
End Enum
```

  – If the type is not specified, **Integer** is used.

  – By default, the first **Enum** member is assigned the value 0, the second member 1, etc.

  – Constant values can be explicitly assigned.

- **You can make use of an enumeration type by declaring a variable of the type indicated in the *Enum* declaration (e.g., *BookingStatus*). You can refer to the enumerated values by using the dot notation.**

```
Dim status As BookingStatus
...
If status = BookingStatus.HotelNotFound Then
   ...
```

 Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Reference Types

- **A variable of a reference type does not directly contain its data, but instead provides a *reference* to the data stored elsewhere (the heap).**
- **In Visual Basic there are the following kinds of reference types:**
  - Class
  - Array
  - Interface
  - Delegate
- **Reference types have a special value *Nothing* which indicates the absence of an instance.**
- **You compare two object instances of a reference type by using the *Is* or *IsNot* operator.**
  - For an illustration, see the example program **DemoCompare**.

```
...
If a Is c Then
   Console.WriteLine("a = c")
End If
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Class Types

- **A class type defines a data structure that has data members, function members, and nested types.**
- **Class types support *inheritance*.**
  - Through inheritance a derived class can extend or specialize a base class.
  - We will discuss inheritance and other details about classes in the next chapter.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Object

- **The *Object* class type is the ultimate base type for all types in Visual Basic.**
  - Every Visual Basic type derives directly or indirectly from **Object**.
- **The *Object* keyword in Visual Basic is an alias for the predefined *System.Object* class.**
- ***System.Object* has methods such as *ToString*, *Equals* and *Finalize*, which we will study later.**
  - Earlier we saw an example of the **GetType** method.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

String Data Type

- **The *String* class also contains many methods and properties that are useful when manipulating strings.**
  - Properties include **Length.**

```
Dim buf As String

If buf.Length = 0 Then
   buf = "Start Here"
End If
```

  - Methods include **Substring**, **Insert**, **Remove,** etc.

```
Dim buf As String = "Microsoft Visual Basic 6.0"

' Displays Visual Basic
Console.WriteLine(buf.Substring(10, 12))

' Removes 6.0 from end of buf
buf = buf.Remove(buf.Length - 4, 4)

' Inserts .NET at end of buf
buf = buf.Insert(buf.Length, ".NET")

' Displays Microsoft Visual Basic.NET
Console.WriteLine(buf)
```

  - Methods also include **ToUpper**, **ToLower**, **Trim**, **TrimStart**, **TrimEnd**, **PadRight,** and **PadLeft**.

```
Dim dataLine As String
dataLine = dataLine.Trim().ToUpper()
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Copying Strings

- **Recall that Visual Basic has value types and reference types.**
  - A value type contains all of its own data.
  - A reference type refers to data stored somewhere else.
- **As a class, *String* is a reference type.**
- **If a reference variable gets copied to another reference variable, both will refer to the same object.**
- **If the object referenced by the second variable is changed, the first variable will also reflect the new value.**

```
Dim s1 As String = "hello"
Dim s2 As String = s1   ' s2 also refers to "hello"
```

- **To provide more predictable program behavior, strings in Visual Basic are *immutable*.**
  - Once assigned a value, the object a string refers to cannot be changed.
  - What you may think of as changing the value of a string is really giving a new reference.

```
Dim s As String s = "bat"
s = s + "man"       ' a new object is created and
                    ' s is assigned to refer to this
                    ' new object
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

StringBuilder Class

- **As we have just discussed, instances of the *String* class are immutable.**
  - As a result, when you manipulate instances of **String**, you are frequently obtaining new **String** instances.
  - Depending on your applications, creating all of these instances may be expensive.
  - The .NET library provides a special class, **StringBuilder** (located in the **System.Text** namespace), in which you may directly manipulate the underlying string without creating a new instance.
  - When you are done, you can create a **String** instance out of an instance of **StringBuilder** by using the **ToString** method.
- **A *StringBuilder* instance has a capacity and a maximum capacity.**
  - These capacities can be specified in a constructor when the instance is created.
  - By default, an empty **StringBuilde**r instance starts out with a capacity of 16.
  - As the stored string expands, the capacity will be increased automatically.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

StringBuilderDemo

- **The program *StringBuilderDemo* provides a simple demonstration of using the *StringBuilder* class.**
  - It shows the starting capacity and the capacity after strings are appended. At the end, a **String** is returned.

```
' StringBuilderDemo.vb

Imports System.Text

Module StringBuilderDemo
   Public Sub Main()
      Dim build As StringBuilder = _
         New StringBuilder()
      Console.WriteLine("capacity = {0}", _
         build.Capacity)
      build.Append("This is the first sentence." _
         + vbLf)
      Console.WriteLine("capacity = {0}", _
         build.Capacity)
      build.Append("This is the second sentence." _
         + vbLf)
      Console.WriteLine("capacity = {0}", _
         build.Capacity)
      build.Append("This is the last sentence." _
         + vbLf)
      Console.WriteLine("capacity = {0}", _
         build.Capacity)
      Dim str As String = build.ToString()
      Console.Write(str)
   End Sub
End Module
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Classes and Structures

- **In Visual Basic the key difference between a class and a structure is that a class is a reference type and a structure is a value type.**
- **A class must be instantiated explicitly using *New*.**
  - The new instance is created on the heap, and memory is managed by the system through a garbage collection process.
- **A structure instance may simply be declared, or you may use *New*.**
  - For a structure the new instance is created on the stack, and the instance will be deallocated when it goes out of scope.
- **There is different semantics for assignment, whether done explicitly or via call by value mechanism in a method call.**
  - For a class you will get a second object reference, and both object references refer to the same data.
  - For a structure you will get a completely independent copy of the data in the structure.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Arrays

- **An array is a collection of elements with the following characteristics.**
  - All array all elements must be of the same type. The element type of an array can be any type, including an array type. An array of arrays is often referred to as a *jagged* array.
  - An array may have one or more dimensions. For example, a two dimensional array can be visualized as a table of values. The number of dimensions is known as the array's *rank*.
  - Array elements are accessed using one or more computed integer values, each of which is known as an *index*. A one-dimensional array has one index.
  - In Visual Basic .NET, an array index starts at 0, as in C, C++, C# and Java.
  - The elements of an array are created when the array object is created. The elements are automatically destroyed when there are no longer any references to the array object.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

One Dimensional Arrays

- **An array is declared using parentheses after the variable.**

```
Dim a() As Integer   ' declares an array of Integer
```

  – Note that the *size* of the array is not part of its type.

  – The variable declared is a *reference* to the array.

- **You create the array elements and establish the size of the array using the *New* operator.**

```
a = New Integer(9){}  ' creates 10 array elements
```

  – The number in the parentheses is the upper bound of the index, which is one less than the size of the array.

- **You can indicate you are done with the array elements by assigning the array reference to *Nothing*.**

```
a = Nothing
```

- **You may both declare and initialize array elements using curly brackets.**

```
Dim primes() as Integer = {2, 3, 5, 7, 11}
```

  – The garbage collector is now free to deallocate the elements.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

System.Array

- **Arrays are objects.**
  - **System.Array** is the abstract base class for all array types.
- **Accordingly you can use the properties and methods of *System.Array* for any array.**

```
Array.Sort(a)         ' sorts the array
Dim i As Integer
For i = 0 To a.Length - 1
    Console.Write("{0} ", a(i))
Next
Console.WriteLine()
```

- **For a sample array program, see *ArrayDemo*.**

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

InputWrapper Class

- **The *InputWrapper* class "wraps" interactive input for several basic data types.**
  - The supported data types are **Integer**, **Double**, **Decimal** and **String**.
  - Methods **getInt**, **getDouble**, **getDecimal** and **getString** are provided.
  - A prompt string is passed as an input parameter.
- **See the directory *TestInputWrapper*.**
  - The file **InputWrapper.vb** implements the class
  - The file **TestInputWrapper.vb** tests the class.
- **You do not need to be familiar with the implementation of *InputWrapper* in order to use it.**
  - That is the beauty of "encapsulation" – complex functionality can be hidden by an easy to use interface.

```
Dim iw As New InputWrapper()
Console.WriteLine("Enter command, quit to exit")
cmd = iw.getString(": ")
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Input Wrapper Implementation

```
' InputWrapper.vb

' Class to wrap simple stream input
' Datatypes supported:
'       Integer
'       Double
'       Decimal
'       String

Class InputWrapper
    Public Function getInt( _
     ByVal prompt As String) As Integer
        Console.Write(prompt)
        Dim buf As String = Console.ReadLine()
        Return Convert.ToInt32(buf)
    End Function
    Public Function getDouble( _
     ByVal prompt As String) As Double
        Console.Write(prompt)
        Dim buf As String = Console.ReadLine()
        Return Convert.ToDouble(buf)
    End Function
    Public Function getDecimal( _
     ByVal prompt As String) As Decimal
        Console.Write(prompt)
        Dim buf As String = Console.ReadLine()
        Return Convert.ToDecimal(buf)
    End Function
    Public Function getString( _
     ByVal prompt As String) As String
        Console.Write(prompt)
        Dim buf As String = Console.ReadLine()
        Return buf
    End Function
End Class
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Jagged Arrays

- **You can declare an array of arrays, or a "jagged" array.**
  - Each row can have a different number of elements.

```
Dim binomial()() As Integer
```

- **You then create the array of rows, specifying how many rows there are (each row is itself an array).**

```
binomial = New Integer(rows - 1)() {}
```

- **Next you create the individual rows:**

```
binomial(i) = New Integer(i) {}
```

- **Finally you can assign individual array elements.**

```
binomial(0)(0) = 1
```

- ***Pascal* program creates and prints Pascal's triangle.**

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

- **Higher dimensional jagged arrays can be created following the same principles.**

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Rectangular Arrays

- **Visual Basic also permits you to define rectangular arrays.**
  - All rows have the same number of elements.
- **First you declare the array:**

```
Dim MultTable(,) As Integer
```

- **Then you create all the array elements, specifying number of rows and columns:**

```
MultTable = New Integer(rows - 1, columns - 1) {}
```

- **Finally you can assign individual array elements.**

```
MultTable(i, j) = i * j
```

- **The *RectangularArray* program creates and prints out a multiplication table.**

```
  0    0    0    0    0
  0    1    2    3    4
  0    2    4    6    8
  0    3    6    9   12
  0    4    8   12   16
```

- **Higher dimensional rectangular arrays can be created following the same principles.**

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

For Each for Arrays

- **Visual Basic provides a *For Each* loop that can be used to iterate through the elements of an array.**
  - The **Pascal** sample code used nested **For Each** loops to print all the elements of a jagged array.

```
Console.WriteLine( _
   "Pascal triangle via nested For Each loop")
Dim row() As Integer
For Each row In binomial
   Dim x As Integer
   For Each x In row
       Console.Write("{0} ", x)
   Next
   Console.WriteLine()
Next
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Boxing and Unboxing

- **One of the strong features of Visual Basic is that is has a unified type system.**
- ***Every* type, including the simple built-in types such as *Integer*, derive from *System.Object*.**
  - In Visual Basic "everything is an object".
- **A language such as Smalltalk also has such a feature, but pays the price of inefficiency for simple types.**
- **Languages such as C++ and Java (before Java 5.0) treat simple built-in types differently than objects, thus obtaining efficiency but at loss of a unified type system.**
- **.NET languages, such as Visual Basic and C#, enjoy the best of both worlds through a process known as "boxing".**
  - "Boxing" converts a value type such as **Integer** or a **Structure** to an object, and is done implicitly.
  - "Unboxing" converts a boxed value type (stored on heap) back to unboxed simple value (stored on stack). Unboxing is done through a type cast.

```
Dim x As Integer = 5
Dim obj As Object = x     ' boxing
x = CInt(obj)             ' unboxing
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Output in Visual Basic

- **Simple output (e.g. for debugging) for various data types can be done using *Console.WriteLine* method applied to a *string*.**
  - The **ToString** method of **System.Object** will provide a string representation for any data type.
  - For custom data types you should override **ToString** (see Chapter 4).
  - You can use the **&** concatenation operator for strings to build up an output string (a technique that can also be applied in other contexts, such as building a SQL query string).

```
Dim x As Integer = 24
Dim y As Integer = 26
Console.WriteLine("Product of " & x & " and " _
   & y & " is " & x * y)
```

- **Alternatively you can use {0}, {1}, etc. as placeholders.**

```
Console.WriteLine( _
   "Product of {0} and {1} is {2}", x, y, x * y)
```

Output:

```
Product of 24 and 26 is 624
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Formatting

- **Visual Basic has extensive formatting capabilities, which you can control through the placeholders.**
  - Simplest: {n} where n is 0, 1, 2, ...
  - Control width: {n,w} and w is width (positive for right justified and negative for left justified)
  - Format string: {n:S} where S is a format string
  - Width and format string: {n,w:S}
- **A format string consists of a format character followed (optionally) by a precision specifier.**

Format Character	Meaning
C	Currency (locale specific)
D	Decimal integer
E	Exponential (scientific)
F	Fixed-point
G	General (E or F)
N	Number with embedded commas
X	Hexadecimal

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Formatting Example

```
Dim pi As Double = Math.PI
Dim cost As Decimal = 70.45D
Console.WriteLine("{0,30}", pi)    ' width 30
Console.WriteLine("{0,-30}", pi)   ' left justified
Console.WriteLine("{0,30:F}", pi)  ' fixed point
Console.WriteLine("{0,30:F4}", pi) ' precision 4
Console.WriteLine("{0,30:C}", cost)' currency
```

Output:

```
            3.1415926535897931
3.1415926535897931
                          3.14
                        3.1416
                        $70.45
```

– See **FormatDemo**.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Modules

- **In Visual Basic, there are several types of code files (or modules), all of which have the *.vb* extension.**
  - A **Class** represents a class (in the object-oriented sense); it contains all attributes and methods of the class. An instance of the class must generally be created before its members may be accessed. We discuss classes in Chapter 3.
  - A **Form** represents a window and all of its associated data, procedures, and event handlers. In Visual Basic, a form is simply a type of class. We discuss forms in Chapter 6.
  - A **Module** represents a collection of "global" data and procedures.
- **Each of the items listed above contain a *declarations* section that can be used to declare structures, constants, and variables.**
  - Items declared in this section are available to all code in the module.
- **These items, and all procedure declarations, may use the word *Public* or *Private* to define its visibility outside of the module.**
  - If the item is declared as **Public**, that means it can be accessed by code outside of the module.
  - If the item is declared as **Private**, that means that it can be accessed only by code within the module.
  - Variables declared using **Dim** are considered private.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Subroutines and Functions

- **Visual Basic supports both subroutines and functions.**
  - *Subroutines* are procedures that do not have an explicit return value.
  - *Functions* are procedures that have an explicit return value.
- **A subroutine can be defined as:**

```
Syntax:

  [visibility] Sub name (argumentList)
      ' statements
  [Exit Sub]
      ' statements
  End Sub
```

- Parameters passed to the subroutine *by value* are denoted **ByVal** and result in a copy being made.
- Parameters passed *by reference* are denoted **ByRef** and can be used for passing data back to the calling procedure.

```
' Calculates shipping costs based on weight
' $12.50 for first 3 lbs plus $3.00 each extra lb
Private Sub CalcCost (ByVal weight as Integer, _
 ByRef amt As Single)
   amt = 0
   If weight < 0 Then Exit Sub
   amt = IIf (weight <= 3, 12.50,_
      12.50 + (weight - 3) * 3)
End Sub
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Subroutines and Functions (Cont'd)

- **Calling a subroutine in Visual Basic is like calling a procedure in most languages:**

```
Dim shippingCost As Single

' Call the subroutine
CalcCost (4, shippingCost)
```

  - NOTICE: VB6 programmers will notice that parentheses are now *required* for parameter lists.

- **A function, in its simplest form, is defined as follows:**

```
Syntax:

  [visibility] Function name (argumentList) _
       [As returnType]
       ' statements
       Return returnValue
  End Function
```

  - Parameters are also passed to the function by value, unless otherwise noted.
  - The return type is assumed to be **Object** unless specified.
  - NOTICE: VB6 programmers should notice that a **Return** statement can now be used to return values from a function. You may also use the VB6 technique of assigning the return value to the name of the function.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Subroutines and Functions (Cont'd)

```
' Calculates shipping costs based on weight
' $12.50 for first 3 lbs plus $3.00 each extra lb
Private Function CalcCost(ByVal weight As _
 Integer) As Single

   Dim cost As Single = 0F
   If weight < 0 Then Exit Function
   cost = IIf (weight <= 3, 12.50, _
      12.50 + (weight - 3) * 3)
   Return cost

End Function
```

- **Calling a function in Visual Basic is like calling a function in most languages:**

```
Dim itemWeight As Integer
Dim shippingCost As Single

' Call the function
shippingCost = CalcCost (itemWeight)
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Default Parameters

- **Visual Basic allows parameters to have default values.**
  - By specifying default values for parameters, they become optional.

```
' Calculates shipping costs; assumes that
' the US Postal Service is the default shipper.
' shipper codes: 1=USPS, 2=FEDEX, 3=UPS

Private Function CalcCost(weight as Integer, _
 Optional ByVal shipper As Integer = 1) As Single

  Dim cost As Single = 0F
  Select Case shipper
  Case 1
    ' calculate cost for USPS
  Case 2
    ' calculate cost for FEDEX
  Case 3
    ' calculate cost for UPS
  End Select
  Return cost

End Function
```

<u>Example of Usage</u>

```
Dim amt As Single

amt = CalcCost(13)
amt = CalcCost(13,2)
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Exceptions

- **Visual Basic provides an exception handling mechanism that is similar to those found in languages such as Java and C++.**
  - Exceptions are implemented by the .NET Framework, so exceptions can be thrown in an assembly written in one .NET language and caught in an assembly written in another.
- **Exception handling is accomplished via the following:**
  - Code that might encounter an exception is enclosed in a **Try** block.
  - Exceptions are caught in one or more **Catch** blocks that immediately follow the **Try**.
  - A match is made based on the data type of the **Exception** object that is passed to the **Catch**.
  - Code that should be executed regardless of whether an exception was encountered is enclosed in a **Finally** block, located immediately below the last **Catch**.
  - The entire block is terminated by an **End Try** statement.
- **Exceptions can be generated by the system or can be specifically thrown with a *Throw* statement.**

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Exceptions – Example Program

- **Several features of exception handling are illustrated in the *ExceptionDemo* program.**
  - User is prompted to enter a positive integer.
  - Input is checked for proper format, but not for being positive. The numbers entered are stored in an array.
  - A count is made of all inputs, properly formatted or not.
  - The contents of the array are displayed by calling a subroutine for each array element. The subroutine throws an exception if the number is negative.

```
Module ExceptionDemo
   Sub Main()
      Dim numList(3) As Integer
      Dim iw As New InputWrapper()
      Dim i As Integer = 0
      Dim count As Integer = 0
      Do Until i >= numList.Length
         Try
            numList(i) = iw.getInt( _
               "Enter positive integer: ")
            i = i + 1
            ' Catching a system generated exception
            ' (probably from bad input)
         Catch e As Exception
            Console.WriteLine( _
               "Error: " & e.Message)
         Finally
            count += 1
         End Try
      Loop
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Exceptions – Example Program

```
      Console.WriteLine( _
         "{0} inputs processed", count)

      Console.WriteLine("Numbers are: ")
      For i = 0 To numList.Length - 1
         Try
            ShowPositiveNumber(numList(i))
         Catch e As Exception
            Console.WriteLine( _
               "Error: " & e.Message)
         End Try
      Next
   End Sub

   Sub ShowPositiveNumber(ByVal num As Integer)
      If num < 0 Then
         Throw New Exception( _
            "Number " & num & " is negative")
      End If
      Console.WriteLine("number = {0}", num)
   End Sub

End Module
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

System.Exception

- **The *System.Exception* class contains many properties and methods that can be used to gather information about an exception.**
  - The **Source** property returns a string containing the name of the application or object that threw the exception.
  - The **TargetSite** property returns a string containing the name of the method that threw the exception.
  - The **Message** property returns a string containing information about the exception.
  - The **StackTrace** property returns a string with stack trace information identifying the point where the exception was encountered.
  - The **InnerException** property returns a reference to another exception that was active when the current exception was thrown.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Implicit Line Continuation

- **Visual Basic 2010 introduced the new feature of *implicit line continuation*.**
- **In many cases you may continue a statement on the next line without using an underscore:**
  - After a comma
  - After an open parentheses or before a closing parentheses
  - After an open curly brace { or before a closing curly brace }
  - After the concatenation operator &
  - After assignment operators =, &=, += and so on
  - After binary operators +, -, *, /, **Mod**, <, >, **And**, **Or** and so on
  - After the **Is** and **IsNot** operators
  - After a member qualifier operator . and before the member name
  - After the **In** keyword in a **For Each** statement
- **For complete rules consult the MSDN documentation.**

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Lab 2

Implementing a Customers Class

In this lab, you will begin the Acme Travel Agency case study by implementing a simple Customers class in Visual Basic. You are provided with starter code that defines a class for an individual customer and a test program. You are to implement a class that can be used by Acme to keep track of customers who register for its services. Customers supply their first and last name, and email address. The system assigns a customer id. The following features are supported:

- Register a customer, returning a customer id
- Unregister a customer
- Obtain customer information, either for a single customer or for all customers (pass the customer id, and for customer id of –1 return all customers)
- Change customer’s email address

Detailed instructions are contained in the Lab 2 write-up at the end of the chapter.

Suggested time: 60 minutes

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Summary

- **Visual Basic is a high-level, object-oriented programming language with a variety of data types and control structures.**
- **Visual Basic supports both subroutines and functions.**
- **The *System* class includes methods for doing input and output, such as *ReadLine* and *WriteLine*.**
- **The .NET Framework has a large class library that is partitioned into namespaces.**
- **Visual Basic has value and reference data types.**
- **Through boxing and unboxing, Visual Basic achieves a unified type system, with all types acting as if they are derived from *Object*.**
- **Built-in numeric types, *Boolean*, and *Structure* are value types.**
- **Examples of reference types are *Object*, *String*, and arrays.**
- **Visual Basic has extensive formatting capabilities, which you can control through the placeholders.**
- **Exceptions in Visual Basic are implemented by the Common Language Runtime, so exceptions can be thrown in one .NET language and caught in another.**

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Lab 2

Implementing a Customers Class

Introduction

In this lab, you will begin the Acme Travel Agency case study by implementing a simple Customers class in Visual Basic. You are provided with starter code that defines a class for an individual customer and a test program. You are to implement a class that can be used by Acme to keep track of customers who register for its services. Customers supply their first and last name, and email address. The system assigns a customer id. The following features are supported:

- Register a customer, returning a customer id
- Unregister a customer
- Obtain customer information, either for a single customer or for all customers (pass the customer id, and for customer id of –1 return all customers)
- Change customer's email address

Suggested Time: 60 minutes

Root Directory: **OIC\VbEss**

Directories:	**Labs\Lab2\Acme**	(Do your work here)
	CaseStudy\Acme\Step0	(Backup of starter files)
	CaseStudy\Acme\Step1	(Answer)
Files:	**Customer.vb**	
	Test.vb	

Instructions

1. Build the starter program. There is a complete implementation of a **Customer** class and a stub implementation of a **Customers** class. There is also a test program. Examine the starter code and run the program. Notice that the test program handles exceptions. For example, the stub **GetCustomer** function returns a **Nothing**, which is checked for in the test program. Also, if you enter non-numeric data when prompted for an id in the test program, an exception will be thrown.

2. Add to the **Customers** class declarations of the following private members: an array **custs** of type **Customer()** and a variable **nextc** of type **Integer**. We will use **nextc** as the index of the next element to be added to the array, and it should be initialized to 0.

3. Add code to the **Customers()** constructor that will instantiate the customers array to have 10 elements and register some sample customers.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

4. Add code in **RegisterCustomer** to instantiate a new **Customer** with the specified fields, store this customer in the array, increment **nextc**, and return the id of this new customer. (Note that an id is automatically generated by the constructor of **Customer**.)

5. Replace the stub code in **GetCustomer** by code that will assign **count** to be **nextc** and return **custs**. (Temporarily, we are trying to always return the entire array.) Build and test. Now you should see your sample data returned in response to the “customers” command. Also, the “register” command should be working, so that you can register additional customers.

6. Now we want to provide the full functionality of **GetCustomer**. If id of –1 is passed, the entire array is passed back. Otherwise, an array of 1 element is created, having the customer information for the id that is provided. To implement this feature, first provide code for the helper method **FindId**. This method does a linear search for the given id. If not found, it returns –1. Otherwise it returns the index at which the id was found.

7. Now finish the implementation of **GetCustomer**. Build and test. Now you should be able to query for a single customer by id, as well as obtain the complete list of customers.

8. Implement **UnregisterCustomer**. If the customer is not found, throw an exception. Otherwise, delete the customer from the array. Move the elements after the deleted element up in the array, to fill the deleted item. Build and test.

9. Finally, implement **ChangeEmailAddress**. Build and test. Your miniature customer management system should now be completely working!

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Chapter 6

Introduction to Windows Forms

 Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Introduction to Windows Forms

Objectives

After completing this unit you will be able to:

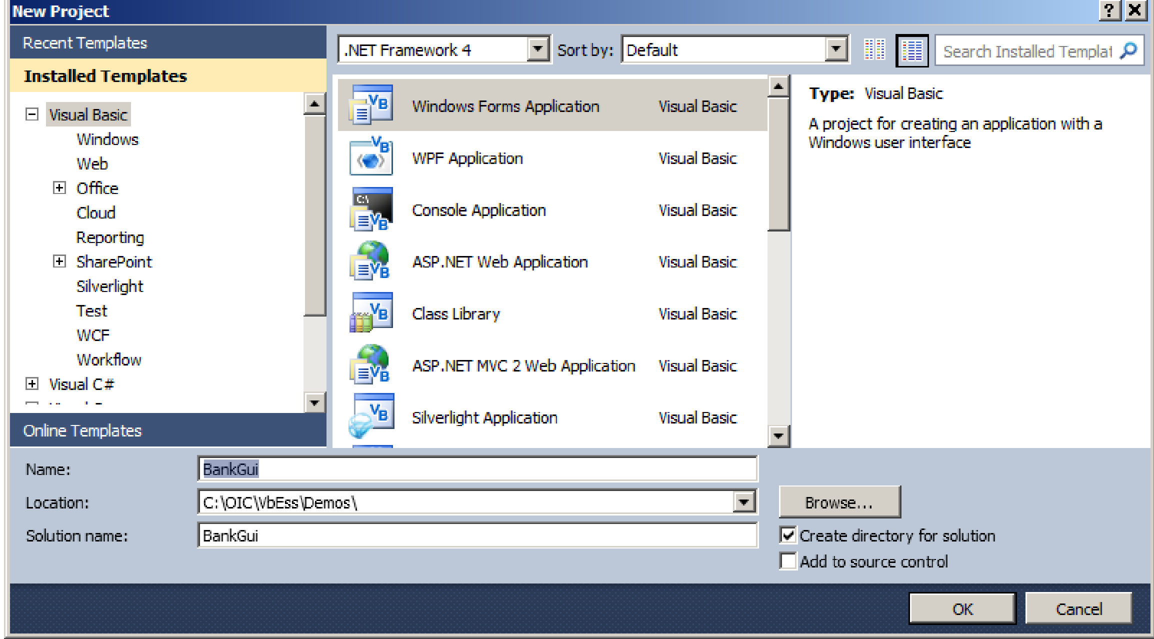
- **Use Visual Studio 2010 to implement a simple graphical user interface with Windows Forms and basic controls.**
- **Use partial classes to organize class code over multiple files.**
- **Explain the principles of event handling in Windows Forms and implement handlers for simple control events.**
- **Use a ListBox control to display a list of objects.**
- **Use the built-in object *My* to access information about the application and its runtime environment.**

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Creating a Windows Forms App

- **It is easy to create a Windows Forms application using the Forms Designer in Visual Studio 2010.**
  - This same Forms Designer is available to all .NET languages.

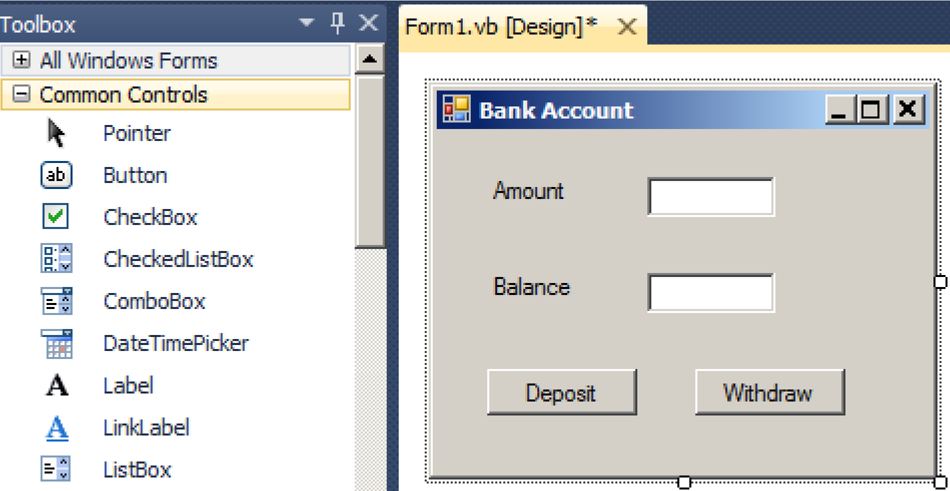
1. Create a new Visual Basic project **BankGui** of type Windows Forms Application. The Location should be the Demos directory. Create a separate directory for the solution.



 Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Windows Forms Demo (Cont'd)

2. From the toolbox, drag two labels, two textboxes, and two buttons to the form. Use the resizing handles to resize the form.



3. Enter property values for the form, textboxes and buttons:

Name	Text
Form1	Bank Account
txtAmount	(blank)
txtBalance	(blank)
cmdDeposit	Deposit
cmdWithdraw	Withdraw

4. Add event handlers for the buttons by double-clicking on each button.

5. Add an event handler for the Form's **Load** event by double-clicking over any empty area of the form.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Windows Forms Demo (Cont'd)

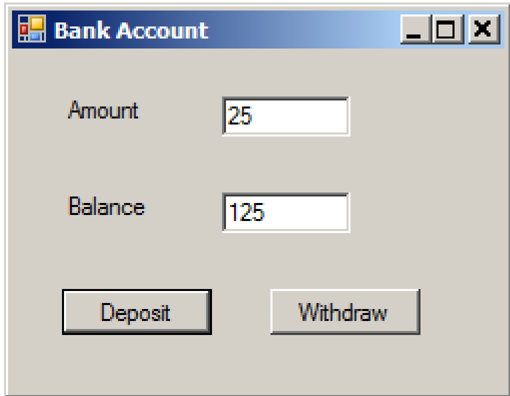
6. Add the following code:

```
Public Class Form1

   Private Sub cmdDeposit_Click(ByVal sender As _
   System.Object, ByVal e As System.EventArgs) _
   Handles cmdDeposit.Click
      Dim amount As Integer = _
         Convert.ToInt32(txtAmount.Text)
      Dim balance As Integer = _
         Convert.ToInt32(txtBalance.Text)
      balance += amount
      txtBalance.Text = Convert.ToString(balance)
   End Sub

   Private Sub cmdWithdraw_Click(ByVal sender As _
   System.Object, ByVal e As System.EventArgs) _
   Handles cmdWithdraw.Click
      Dim amount As Integer = _
         Convert.ToInt32(txtAmount.Text)
      Dim balance As Integer = _
         Convert.ToInt32(txtBalance.Text)
      balance -= amount
      txtBalance.Text = Convert.ToString(balance)
   End Sub

   Private Sub Form1_Load(ByVal sender As _
   System.Object, ByVal e As System.EventArgs) _
   Handles MyBase.Load
      txtAmount.Text = "25"
      txtBalance.Text = "100"
   End Sub
End Class
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Windows Forms Demo (Cont'd)

7. Build and run the application. It should behave like a standard Windows application.



– The completed demo is in the **BankGui\Step1** directory for this chapter.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Partial Classes

- **Beginning with .NET 2.0, you can split the definition of a class over two or more source files.**
  - Use the **Partial** keyword modifier.
  - Code maintenance can be simplified for large classes.
- **The Windows Forms Designer generates partial classes.**
  - The boiler plate code generated by the Forms Designer is kept in one file, which is not touched by the programmers.
  - Code added by the programmer is maintained in another file.

```
' Form1.Designer.vb
Partial Class Form1
    Inherits System.Windows.Forms.Form

...
    Private Sub InitializeComponent()
      Me.cmdWithdraw = _
         New System.Windows.Forms.Button
      Me.cmdDeposit = _
         New System.Windows.Forms.Button
      Me.txtBalance = _
         New System.Windows.Forms.TextBox
      ...
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Windows Forms Event Handling

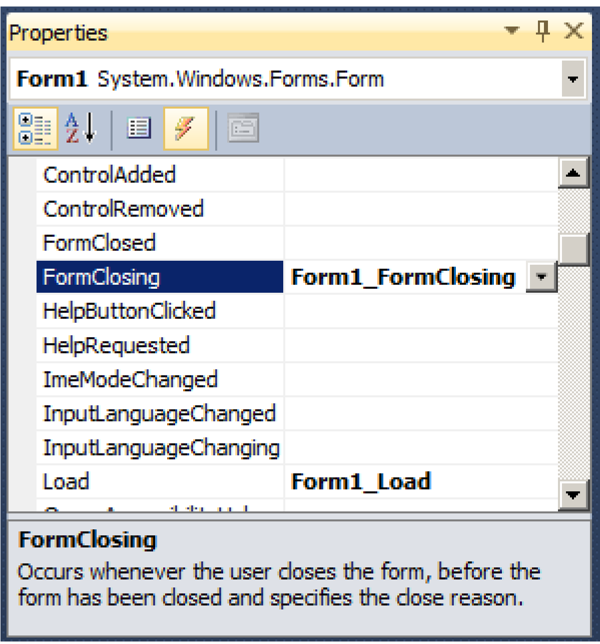
- **GUI applications are event-driven, in which the application executes code in response to user events, such as clicking the mouse, choosing a menu item, etc.**
- **Each form or control has a predefined set of events.**
  - For example, every button has a **Load** event.
- **Windows Forms uses the .NET *event* model, which uses *delegates* to bind events to the methods that handle them.**
  - When an event occurs in an application, the control raises the event by calling the delegate for that event.
  - The delegate then calls all of the methods it is bound to.
- **The Forms Designer code uses static event handling**

```
' In Form1.Designer.vb

Private WithEvents cmdWithdraw _
   As System.Windows.Forms.Button


' In Form1.vb

Private Sub cmdWithdraw_Click(ByVal sender As _
System.Object, ByVal e As System.EventArgs) _
Handles cmdWithdraw.Click
    ...
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Add Events for a Control

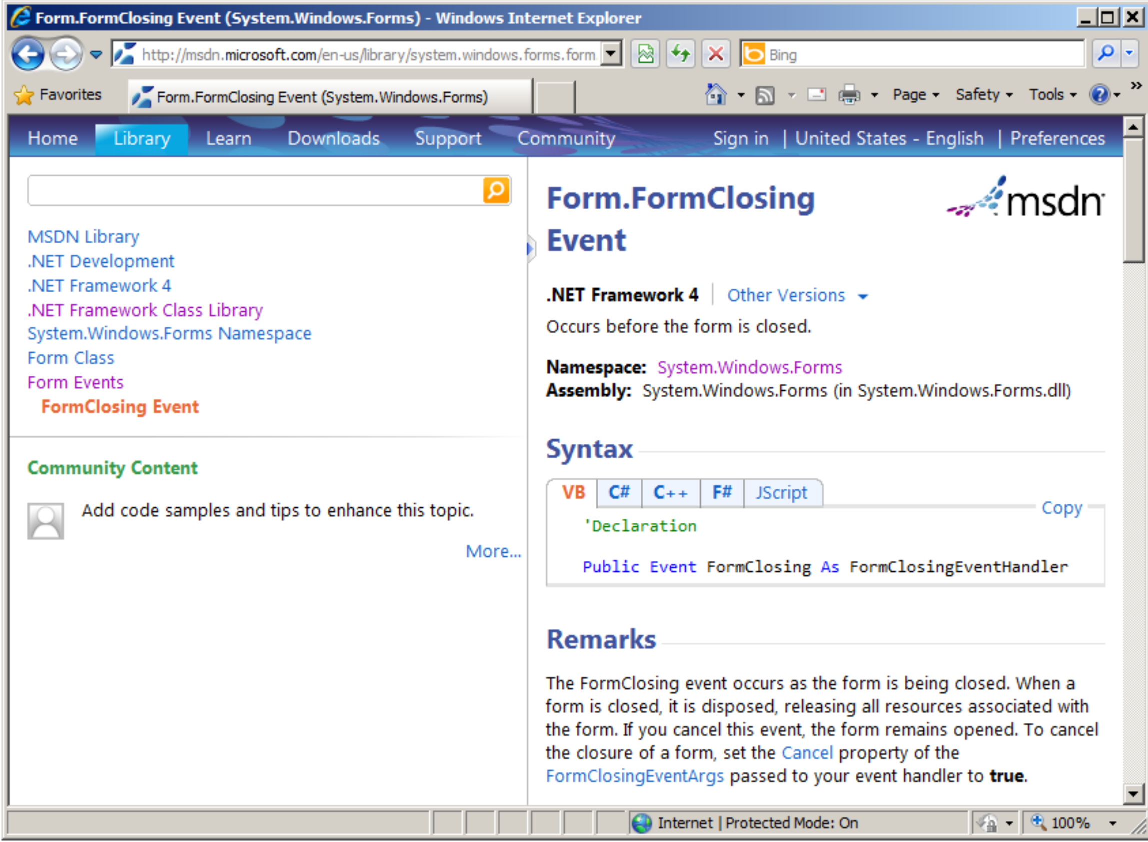
- **Every control has a *primary* event. You can add a handler for this event by double-clicking on the control in the Visual Studio Form Designer.**
  - We added a handler for the Deposit and Withdraw buttons in this way.
- **You can add other events by selecting the in the Properties window and double-clicking on a selected event.**
  - This technique is the same as in Visual C# and is new for Visual Basic beginning with Visual Studio 2005.



Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Events Documentation

- **You can find all of the events associated with a class in the .NET Framework Reference.**
  - This screen shot shows documentation for the **FormClosing** event of the **Form** class.



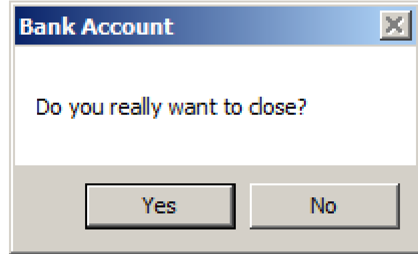
Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Closing a Form

- **A form may be closed in a number of ways, such as by clicking the ☒ at the top-right, using the system menu at the top-left, pressing Alt+F4, and so on.**
- **In all cases, the *FormClosing* event will be fired, providing an opportunity to cancel the closing.**
- **Continuing our simple Bank Account form demo, add the following code to the handler of the *FormClosing* event.**
  - See **BankGui\Step2**.

```
Private Sub Form1_FormClosing( _
ByVal sender As Object, ByVal e As
System.Windows.Forms.FormClosingEventArgs) _
Handles Me.FormClosing
   Dim result As DialogResult = MessageBox.Show( _
      "Do you really want to close?", _
      "Bank Account", MessageBoxButtons.YesNo)
   If result <> DialogResult.Yes Then
      e.Cancel = True
   End If
End Sub
```

- **The following dialog will be displayed:**



Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

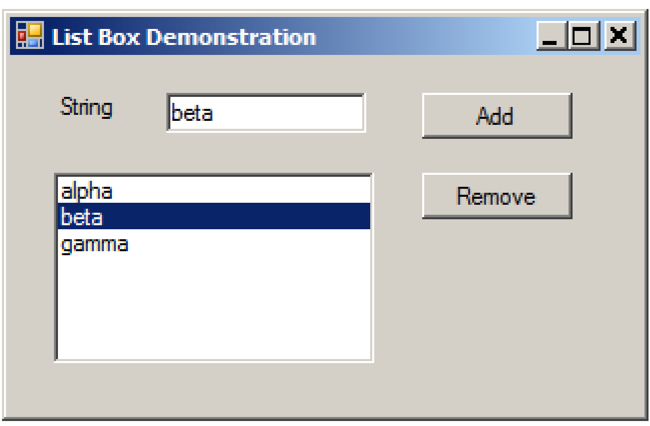
ListBox Control

- **The .NET Framework provides a number of controls that you can use to display lists of items to the user.**
- **These controls also allow the user to select an item from the list, typically by clicking on the item to be selected.**
- **The simplest list control is the *ListBox* control.**
  - The collection **Items** maintains a collection of object references. There are methods such as **Add**, **Remove** and **Clear**.
  - The property **SelectedItem** gets or sets the currently selected item in the listbox. Perform a cast to get out an item of a particular type.
  - The property **SelectedIndex** gets or sets the zero-based index of the currently selected item (–1 if no item is selected).
  - The event **SelectedIndexChanged** is fired when the user selects a new index.
  - There are many more properties, methods, and events, which you can study from the documentation.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

ListBox Example

- **The program *ListBoxDemo* illustrates maintaining a list of strings in a listbox.**

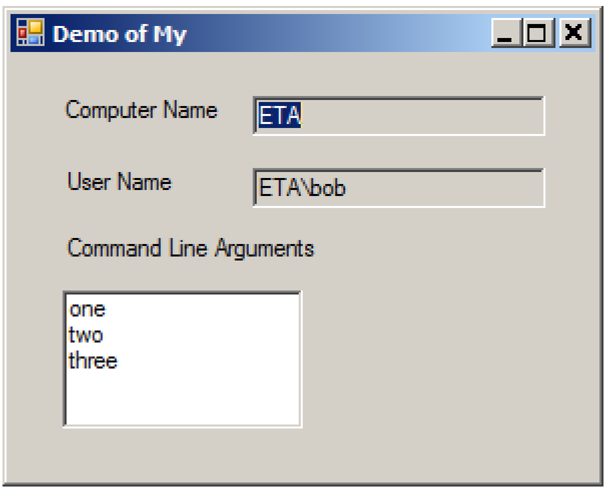



```
Private Sub cmdAdd_Click(ByVal sender As _
System.Object, ByVal e As System.EventArgs) _
Handles cmdAdd.Click
   listBox1.Items.Add(txtString.Text)
End Sub

Private Sub cmdRemove_Click(ByVal sender As _
System.Object, ByVal e As System.EventArgs) _
Handles cmdRemove.Click
   listBox1.Items.Remove(txtString.Text)
End Sub

Private Sub listBox1_SelectedIndexChanged( _
ByVal sender As System.Object, _
ByVal e As System.EventArgs) _
Handles listBox1.SelectedIndexChanged
   If listBox1.SelectedIndex <> -1 Then
      txtString.Text = CStr(listBox1.SelectedItem)
   End If
End Sub
```

 Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

My

- **Visual Basic 2010 provides a built-in object *My* that makes it easy to access much information about the application and its runtime environment.**
  - Important top-level members of **My** include **Computer**, **User**, and **Application**.
  - The example program **MyDemo** provides an illustration, by displaying some information when the form is loaded.



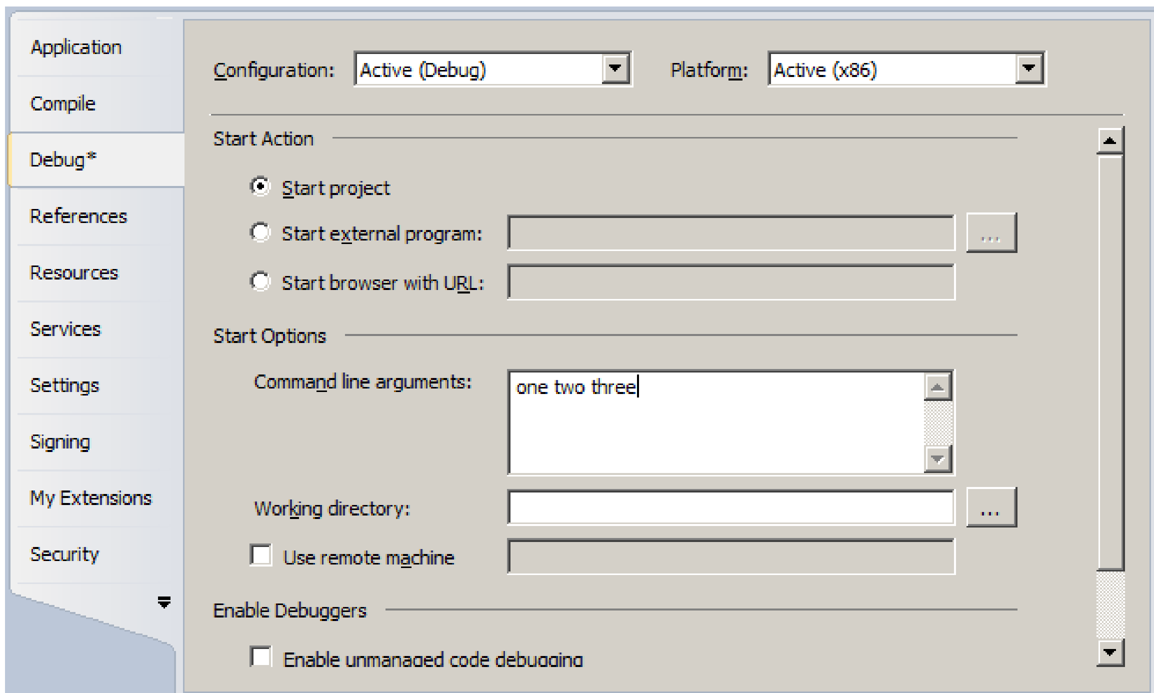
  - Here is the code:

```
Private Sub Form1_Load(ByVal sender As _
System.Object, ByVal e As System.EventArgs) _
Handles MyBase.Load

   txtComputerName.Text = My.Computer.Name
   txtUserName.Text = My.User.Name
   For Each s As String In _
    My.Application.CommandLineArgs
       lstCmd.Items.Add(s)
   Next

End Sub
```


Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Command Line Arguments

- **You can specify command line arguments in Visual Studio.**
  - Right-click over the project and select Properties.
  - Select the Debug tab.
  - Enter desired command line arguments under the Start Options.



- **To save the command line arguments as part of your project information, be sure not to delete the *.vbproj.user* file.**

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Lab 6

A GUI for Acme

In this lab, you will implement a GUI interface for the Acme customer management system implemented in the previous chapter.

Detailed instructions are contained in the Lab 6 write-up at the end of the chapter.

Suggested time: 60 minutes

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Summary

- **With Visual Studio 2010, you can implement a simple graphical user interface with Windows Forms and basic controls.**
- **You can use partial classes to organize class code over multiple files.**
- **Windows Forms uses the .NET event architecture to handle user interface events.**
- **.NET provides various list controls, such as ListBox, for displaying lists of objects.**
- **With the built-in object *My,* you can easily access information about the application and its runtime environment.**

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Lab 6

A GUI for Acme

Introduction

In this lab, you will implement a GUI interface for the Acme customer management system implemented in the previous chapter.

Suggested Time: 60 minutes

Root Directory: **OIC\VbEss**

Directories:	**Labs\Lab6\AcmeGui**	(do your work here)
	CaseStudy\Acme\Step2C	(previous version)
	CaseStudy\Acme\Step3	(answer)

Files: **AcmeCms.vb**

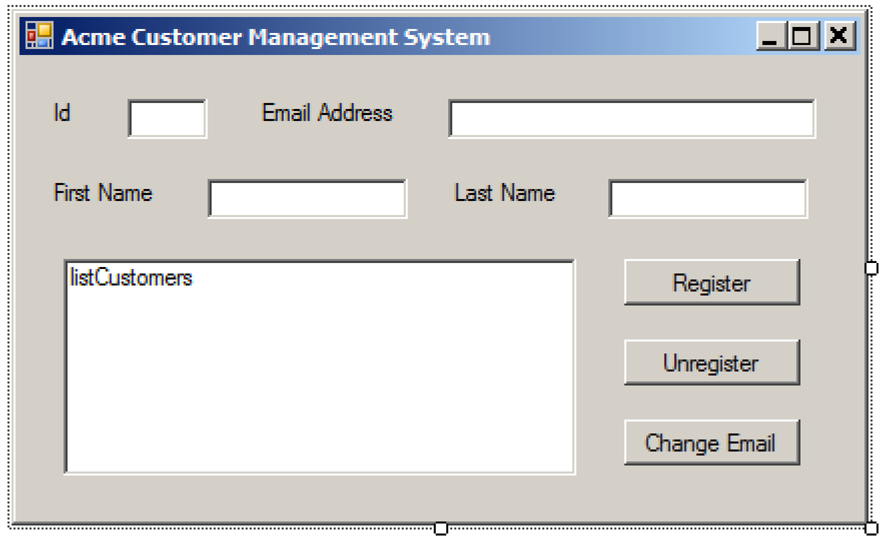
Part A. Create the GUI and List Customers

1. Create a new Visual Basic project of type Windows Application **AcmeGui** located in **Labs\Lab6**.

2. Change the file name **Form1.vb** to **AcmeCms.vb**. The form class will now also be **AcmeCms**.

3. Change the **Text** property of the form to "Acme Customer Management System."

4. Add controls to the form as shown:



Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

5. Make the names of the controls shown: **txtId**, **txtEmailAddress**, **txtFirstName**, **txtLastName**, **listCustomers, cmdRegister, cmdUnregister, cmdChangeEmail.**

6. Copy the files **Customer.vb** and **Test.vb** from **CaseStudy\Acme\Step2C** to the working directory. Add the file **Customer.vb** to the project. (**Test.vb** will be a miscellaneous file that you can refer to as you migrate command-line client code to GUI client code.)

7. Add a private data member **custs** of type **Customers** to class **AcmeCms**.

8. Create a handler for the **Load** event in the **AcmeCms** class. Add code to instantiate a **custs** object.

9. From the file **Test.vb,** copy code to display the customer list by calling a helper method **ShowCustomerArray**, passing it the results of calling the **GetCustomer** method.

10. Implement a private method **ShowCustomerArray** to display the customers in the array list in the list box of the main form. Base your code on the code in **Test.vb**. Don't worry about padding to make neat columns, because it won't work anyway in a listbox with a variable pitch font. (To make neat columns you should use a more advanced control, such as **ListView** or **DataGridView**.) Just use a single space as a separator of fields when you build up a string to add to the listbox. To add a string to a listbox, you can use the following code:

    ```
    listCustomers.Items.Add(str)
    ```

    If the array list is **Nothing**, throw an exception.

11. Build and test the program. You should see a starting list of customers displayed.

Part B. Register a New Customer

In this part, you will add code for registering a new customer.

1. Add an event handler for clicking the Register button.

2. Add code in this handler to call the **RegisterCustomer** method of the **Customers** class. You will pass in the first name, last name, and email address from the textboxes. You will get back an ID, which you will display.

3. Add code to redisplay all of the customers in the listbox. Build and test.

4. You will see the new customer that was added, but you will also see a duplicate of the original customers. Fix this problem by clearing the listbox in the helper method before doing the display. Build and test.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved

Part C. Select a Customer and Split into Fields

In this part, you will add code to select a customer by clicking in the listbox. Relying on the specification that there are no embedded spaces in any of the fields of a customer, you will extract the individual fields using a space as a separator. This extracted information will be shown in the textboxes.

1. Add a handler for the event **SelectedIndexChanged** of the listbox. (Recall that you use the button to obtain a list of events associated with a control.)

2. Test to see if an item has been selected by checking if **listCustomers.SelectedIndex** is different from –1.

3. If an item has been selected, obtain the selected string and split it into fields (relying on a space as a separator) by the following code:

```
Dim selected As String = CStr(listCustomers.SelectedItem)
Dim sep As Char() = New Char() {" "c}
Dim fields As String()
fields = selected.Split(sep)
```

5. Add code to set the textboxes to the corresponding fields of the selected customer.

6. Build and test. As you select different customers, you should see the information in the textboxes change appropriately.

Part D. Unregister a Customer and Change Email Address

In this final part, you will add code for unregistering a new customer. You will also add code for changing an email address.

1. Add a handler for the Unregister button.

2. Add code to unregister the currently selected customer (if there is one), as shown by the ID. When done, clear the textboxes and refresh the listbox. You may build and test at this point.

3. Add a handler for the Change Email button. Add code to change the email address of the currently selected customer (if there is one).

4. Build and test. Your miniature customer management system should now be fully operational! You are now at Step 3.

Copyright © 2010 Object Innovations Enterprises, LLC
All Rights Reserved