Object-Oriented Programming in Visual Basic

Student Guide
Revision 4.5
Object-Oriented Programming in Visual Basic
Rev. 4.5

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 trademark of Object Innovations.

Authors: Robert J. Oberg and Dana Wyatt

Special Thanks: Ernani Cecon, Jr.

Copyright ©2013 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)

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.NET: What You Need to Know</td>
</tr>
<tr>
<td>2</td>
<td>Fundamentals of Visual Basic Programming</td>
</tr>
<tr>
<td>3</td>
<td>Control Structures</td>
</tr>
<tr>
<td>4</td>
<td>Procedures</td>
</tr>
<tr>
<td>5</td>
<td>Advanced Data Types</td>
</tr>
<tr>
<td>6</td>
<td>Exception Handling</td>
</tr>
<tr>
<td>7</td>
<td>Object-Oriented Programming</td>
</tr>
<tr>
<td>8</td>
<td>Inheritance</td>
</tr>
<tr>
<td>9</td>
<td>Interfaces and Collections</td>
</tr>
<tr>
<td>10</td>
<td>Introduction to Windows Forms</td>
</tr>
<tr>
<td>11</td>
<td>Windows Forms Controls</td>
</tr>
<tr>
<td>12</td>
<td>User Interface Features</td>
</tr>
<tr>
<td>13</td>
<td>Database Programming</td>
</tr>
<tr>
<td>14</td>
<td>Newer Features in Visual Basic</td>
</tr>
<tr>
<td>Appendix A</td>
<td>Using Visual Studio 2012</td>
</tr>
<tr>
<td>Appendix B</td>
<td>Learning Resources</td>
</tr>
</tbody>
</table>
Directory Structure

- The course software installs to the root directory $C:\OIC\IntroVb$.
  - Example programs for each chapter are in named subdirectories of chapter directories Chap01, Chap02, and so on.
  - 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.

- Data files install to the directory $C:\OIC\Data$. 
# Table of Contents (Detailed)

**Chapter 1 .NET: What You Need to Know**
- Getting Started ................................................................. 1
- .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
- Visual Studio 2012 ............................................................. 10
- Creating a Console Application ......................................... 11
- Visual Studio Solutions ..................................................... 12
- Starter Code ........................................................................ 13
- Using the Visual Studio Text Editor ................................. 14
- IntelliSense ......................................................................... 15
- Build and Run the Project ............................................... 16
- Pausing the Output ......................................................... 17
- Visual Basic and GUI Programs ......................................... 18
- .NET Documentation ........................................................ 19
- Summary ............................................................................. 20

**Chapter 2 Fundamentals of Visual Basic Programming**
- Visual Basic ......................................................................... 21
- Hello, World ......................................................................... 23
- Compiling, Running (Command Line) ............................... 25
- Program Structure ............................................................. 26
- Namespaces .......................................................................... 28
- Project Imports ................................................................. 29
- Startup Object ...................................................................... 30
- Naming Standards ............................................................. 31
- Keywords ............................................................................. 32
- Multiple-File Program Structure ......................................... 33
- Using Procedures ............................................................... 34
- Data Types ........................................................................... 35
- IEEE 754 Floating Point Standard ....................................... 36
- Data Type Ranges ............................................................... 38
- Literals ................................................................................ 39
- Variables .............................................................................. 41
- Initialization of Variables .................................................... 43
- Type Checking ..................................................................... 44
- Weak Type Checking Example .......................................... 45
- Constants ............................................................................ 47
- Data Conversions ............................................................... 48
- Operators and Expressions ............................................... 50
## Chapter 3 Control Structures ............................................................... 73

- Control Structures ............................................................................. 75
- If/Then Statement ................................................................................ 76
- If/Then/Else Statement ....................................................................... 77
- Payroll Example ................................................................................... 79
- Select Case Statement .......................................................................... 81
- Lab 3A .................................................................................................... 82
- Looping .................................................................................................. 83
- Do/Loop Statement ................................................................................ 84
- Leap Year Example ............................................................................... 86
- Exiting Do Loops Prematurely ............................................................... 88
- While Statement .................................................................................... 89
- For/Next Statement ............................................................................... 90
- A Random Number Example ............................................................... 91
- Lab 3B .................................................................................................... 94
- Continue Statement ............................................................................... 95
- Summary ............................................................................................... 96

## Chapter 4 Procedures ........................................................................... 99

- Modules ................................................................................................. 101
- Subroutines ........................................................................................... 102
- ByVal Parameters .................................................................................. 103
- ByRef Parameters .................................................................................. 105
- Functions ............................................................................................... 106
- A Game Example ................................................................................... 108
- Access, Modules, and Scope ................................................................. 113
- Scope .................................................................................................... 114
- Scope Example – Part 1 ......................................................................... 115
- Scope Example – Part 2 ......................................................................... 116
- Scope Example – Part 3 ......................................................................... 117
- Static Variables ..................................................................................... 118
- Overloading ......................................................................................... 119
Chapter 7 Object-Oriented Programming ............................................................... 205
Object-Oriented Programming ........................................................................... 207
Objects in Software ............................................................................................. 208
Abstraction and Encapsulation ............................................................................. 209
Classes .................................................................................................................. 210
Creating a Class .................................................................................................... 211
Defining Data Members ......................................................................................... 212
Defining Methods ................................................................................................. 213
Creating Objects .................................................................................................. 215
Using Objects ........................................................................................................ 216
Assigning Object References .............................................................................. 217
Garbage Collection ............................................................................................... 219
Me ......................................................................................................................... 220
Properties ............................................................................................................. 221
Defining Properties ............................................................................................... 222
ReadOnly Properties .............................................................................................. 224
WriteOnly Properties ............................................................................................. 225
Auto-Implemented Properties ............................................................................. 226
Auto-Implemented Property Example .................................................................. 227
Shared Attributes .................................................................................................. 228
Shared Methods and Properties .......................................................................... 230
An Arithmetic Calculator Example ...................................................................... 231
Constructors and Initialization ............................................................................. 233
Defining Constructors ........................................................................................... 234
Default Constructor .............................................................................................. 237
Shared Constructor ............................................................................................... 239
Constructor Examples ........................................................................................... 240
ReadOnly Members .............................................................................................. 241
Constant Members ............................................................................................... 242
Events .................................................................................................................... 243
Defining Events .................................................................................................... 244
Restrictions on Event Procedures ....................................................................... 245
Raising Events ....................................................................................................... 246
Trapping Events Using WithEvents ................................................................... 247
Trapping Events Dynamically ............................................................................. 249
A Banking Example ............................................................................................... 251
ToString() in User-Defined Classes ................................................................... 254
Operator Overloading ......................................................................................... 256
Lab 7 ...................................................................................................................... 258
Chapter 8 Inheritance................................................................................................. 263
Inheritance.............................................................................................................. 265
Inheritance Hierarchy.............................................................................................. 266
Defining a Derived Class ......................................................................................... 267
A Simple Example: Types of People ........................................................................ 268
.NET’s Object Class.................................................................................................. 272
Controlling Base Class Construction ..................................................................... 273
Derived Class Construction ...................................................................................... 274
Derived Class Constructors ...................................................................................... 275
Need For Protected Access Modifier ....................................................................... 277
Access Control.......................................................................................................... 278
Using Protected Access............................................................................................ 279
Exceptions................................................................................................................ 282
Defining Custom Exceptions ..................................................................................... 283
Using Custom Exceptions ........................................................................................ 284
Shadowing Base Class Methods .............................................................................. 285
Polymorphism........................................................................................................... 289
Defining a Method as Overridable .......................................................................... 291
Overriding a Method................................................................................................ 292
Using Heterogeneous Collections with Polymorphic Methods ................................ 293
Abstract Classes....................................................................................................... 295
Not Inheritable Classes ............................................................................................ 297
Type Conversions in Inheritance ............................................................................ 298
CType().................................................................................................................... 300
Lab 8 ......................................................................................................................... 301
Summary.................................................................................................................. 302

Chapter 9 Interfaces and Collections........................................................................ 307
Components and OO in Visual Basic ......................................................................... 309
Interfaces.................................................................................................................. 310
Interfaces in Visual Basic .......................................................................................... 312
Implementing an Interface ....................................................................................... 313
Using an Interface ................................................................................................... 314
Multiple Interfaces .................................................................................................. 315
Using Multiple Interfaces ......................................................................................... 318
TypeOf ... Is and Dynamic Interfaces ...................................................................... 320
Interfaces in Visual Basic and COM ........................................................................ 322
Resolving Ambiguity in Interfaces ........................................................................... 323
.NET Interfaces....................................................................................................... 326
Arrays of User-Defined Objects .............................................................................. 327
Implementing IComparable ..................................................................................... 329
Lab 9A ....................................................................................................................... 331
Collections ............................................................................................................... 332
ArrayList................................................................................................................... 333
ArrayList Methods.................................................................................................... 334
Example: StringList .................................................................................................. 335
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 10 Introduction to Windows Forms</td>
<td>359</td>
</tr>
<tr>
<td>Windows Forms</td>
<td>361</td>
</tr>
<tr>
<td>Creating a Windows Forms App</td>
<td>362</td>
</tr>
<tr>
<td>Aligning Controls</td>
<td>366</td>
</tr>
<tr>
<td>Setting the Tab Order</td>
<td>367</td>
</tr>
<tr>
<td>Partial Classes</td>
<td>368</td>
</tr>
<tr>
<td>Windows Forms Event Handling</td>
<td>369</td>
</tr>
<tr>
<td>Add Events for a Control</td>
<td>370</td>
</tr>
<tr>
<td>Events Documentation</td>
<td>371</td>
</tr>
<tr>
<td>Closing a Form</td>
<td>372</td>
</tr>
<tr>
<td>ListBox Control</td>
<td>373</td>
</tr>
<tr>
<td>ListBox Example</td>
<td>374</td>
</tr>
<tr>
<td>My</td>
<td>375</td>
</tr>
<tr>
<td>Command Line Arguments</td>
<td>376</td>
</tr>
<tr>
<td>Lab 10</td>
<td>377</td>
</tr>
<tr>
<td>Summary</td>
<td>378</td>
</tr>
<tr>
<td>Chapter 11 Windows Forms Controls</td>
<td>383</td>
</tr>
<tr>
<td>Common Properties</td>
<td>385</td>
</tr>
<tr>
<td>Common Events</td>
<td>387</td>
</tr>
<tr>
<td>Event Handling</td>
<td>388</td>
</tr>
<tr>
<td>Sample Controls</td>
<td>389</td>
</tr>
<tr>
<td>Using a Label</td>
<td>390</td>
</tr>
<tr>
<td>Using Mnemonics</td>
<td>391</td>
</tr>
<tr>
<td>Using a Text Box</td>
<td>392</td>
</tr>
<tr>
<td>Using a Button</td>
<td>394</td>
</tr>
<tr>
<td>Lab 11A</td>
<td>395</td>
</tr>
<tr>
<td>Using a Radio Button and Group Box</td>
<td>396</td>
</tr>
<tr>
<td>Using a Check Box</td>
<td>399</td>
</tr>
</tbody>
</table>
**Chapter 12  User Interface Features**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dialog Boxes</td>
<td>425</td>
</tr>
<tr>
<td>MessageBox</td>
<td>427</td>
</tr>
<tr>
<td>Custom Dialogs</td>
<td>428</td>
</tr>
<tr>
<td>Adding a Custom Dialog</td>
<td>430</td>
</tr>
<tr>
<td>Using DialogResult</td>
<td>431</td>
</tr>
<tr>
<td>Displaying a Dialog</td>
<td>433</td>
</tr>
<tr>
<td>Tab Order and Focus</td>
<td>434</td>
</tr>
<tr>
<td>Initializing a Custom Dialog</td>
<td>435</td>
</tr>
<tr>
<td>Changing the Behavior of a Button’s DialogResult</td>
<td>436</td>
</tr>
<tr>
<td>Lab 12A</td>
<td>438</td>
</tr>
<tr>
<td>Modeless Dialogs</td>
<td>439</td>
</tr>
<tr>
<td>Managing the Relationship between Forms</td>
<td>440</td>
</tr>
<tr>
<td>Programming the Apply and Close Buttons</td>
<td>441</td>
</tr>
<tr>
<td>Enabling / Disabling the Apply Button</td>
<td>442</td>
</tr>
<tr>
<td>Preventing Multiple Instances</td>
<td>443</td>
</tr>
<tr>
<td>Common Dialogs</td>
<td>444</td>
</tr>
<tr>
<td>Using a Common Dialog Control</td>
<td>445</td>
</tr>
<tr>
<td>Menus</td>
<td>446</td>
</tr>
<tr>
<td>MenuStrip Control</td>
<td>447</td>
</tr>
<tr>
<td>A Calculator Example Application</td>
<td>448</td>
</tr>
<tr>
<td>Calculator Starter Code</td>
<td>449</td>
</tr>
<tr>
<td>Attaching a Menu to a Form</td>
<td>450</td>
</tr>
<tr>
<td>Configuring Items in a Menu</td>
<td>451</td>
</tr>
<tr>
<td>Testing the Menu</td>
<td>452</td>
</tr>
<tr>
<td>Responding to Menu Events</td>
<td>453</td>
</tr>
<tr>
<td>Changing Menu Item Appearance</td>
<td>454</td>
</tr>
<tr>
<td>Modified SetOperation() Method</td>
<td>455</td>
</tr>
<tr>
<td>ContextMenuStrip Control</td>
<td>456</td>
</tr>
<tr>
<td>Example: Integrating a Context Menu into an Application</td>
<td>457</td>
</tr>
<tr>
<td>Context Menu Events</td>
<td>458</td>
</tr>
<tr>
<td>Handling Multiple Events</td>
<td>459</td>
</tr>
<tr>
<td>Lab 12B</td>
<td>460</td>
</tr>
<tr>
<td>Summary</td>
<td>461</td>
</tr>
<tr>
<td>Using the ToolTip Control</td>
<td>462</td>
</tr>
<tr>
<td>Lab 11B</td>
<td>463</td>
</tr>
<tr>
<td>Using a ListBox Control</td>
<td>464</td>
</tr>
<tr>
<td>Adding ListBox Items</td>
<td>465</td>
</tr>
<tr>
<td>Removing ListBox Items</td>
<td>466</td>
</tr>
<tr>
<td>Selecting an Item in a List Box</td>
<td>467</td>
</tr>
<tr>
<td>Using a ComboBox Control</td>
<td>468</td>
</tr>
<tr>
<td>Flexible Events Handlers</td>
<td>469</td>
</tr>
<tr>
<td>Timer Control</td>
<td>470</td>
</tr>
<tr>
<td>Lab 11C</td>
<td>471</td>
</tr>
<tr>
<td>Using a Common Dialog Control</td>
<td>472</td>
</tr>
<tr>
<td>Using DialogResult</td>
<td>473</td>
</tr>
<tr>
<td>Using DialogResult</td>
<td>474</td>
</tr>
<tr>
<td>Displaying a Dialog</td>
<td>475</td>
</tr>
<tr>
<td>Tab Order and Focus</td>
<td>476</td>
</tr>
<tr>
<td>Initializing a Custom Dialog</td>
<td>477</td>
</tr>
<tr>
<td>Changing the Behavior of a Button’s DialogResult</td>
<td>478</td>
</tr>
<tr>
<td>Lab 12A</td>
<td>479</td>
</tr>
<tr>
<td>Modeless Dialogs</td>
<td>480</td>
</tr>
<tr>
<td>Managing the Relationship between Forms</td>
<td>481</td>
</tr>
<tr>
<td>Programming the Apply and Close Buttons</td>
<td>482</td>
</tr>
<tr>
<td>Enabling / Disabling the Apply Button</td>
<td>483</td>
</tr>
<tr>
<td>Preventing Multiple Instances</td>
<td>484</td>
</tr>
<tr>
<td>Common Dialogs</td>
<td>485</td>
</tr>
<tr>
<td>Using a Common Dialog Control</td>
<td>486</td>
</tr>
<tr>
<td>Menus</td>
<td>487</td>
</tr>
<tr>
<td>MenuStrip Control</td>
<td>488</td>
</tr>
<tr>
<td>A Calculator Example Application</td>
<td>489</td>
</tr>
<tr>
<td>Calculator Starter Code</td>
<td>490</td>
</tr>
<tr>
<td>Attaching a Menu to a Form</td>
<td>491</td>
</tr>
<tr>
<td>Configuring Items in a Menu</td>
<td>492</td>
</tr>
<tr>
<td>Testing the Menu</td>
<td>493</td>
</tr>
<tr>
<td>Responding to Menu Events</td>
<td>494</td>
</tr>
<tr>
<td>Changing Menu Item Appearance</td>
<td>495</td>
</tr>
<tr>
<td>Modified SetOperation() Method</td>
<td>496</td>
</tr>
<tr>
<td>ContextMenuStrip Control</td>
<td>497</td>
</tr>
<tr>
<td>Example: Integrating a Context Menu into an Application</td>
<td>498</td>
</tr>
<tr>
<td>Context Menu Events</td>
<td>499</td>
</tr>
<tr>
<td>Handling Multiple Events</td>
<td>500</td>
</tr>
<tr>
<td>Lab 11B</td>
<td>501</td>
</tr>
<tr>
<td>Summary</td>
<td>502</td>
</tr>
</tbody>
</table>
Chapter 13 Database Programming................................................................. 473
  ADO.NET ........................................................................................................ 475
  ADO.NET Architecture .................................................................................... 476
  .NET Data Providers ......................................................................................... 478
  Programming with ADO.NET Interfaces .......................................................... 479
  .NET Namespaces .............................................................................................. 480
  Connected Data Access ...................................................................................... 481
  Sample Database ............................................................................................... 482
  Example: Connecting to SQL Server ............................................................... 483
  ADO.NET Class Libraries .................................................................................. 484
  Connecting to an OLE DB Data Provider .......................................................... 485
  Importing a Namespace ...................................................................................... 486
  Using Commands ............................................................................................... 487
  Creating a Command Object ............................................................................. 488
  ExecuteNonQuery .............................................................................................. 489
  Using a Data Reader ......................................................................................... 490
  Data Reader: Code Example ............................................................................ 491
  Lab 13A ............................................................................................................. 492
  Disconnected Datasets ...................................................................................... 493
  Data Adapters .................................................................................................... 494
  Data Bound Controls ....................................................................................... 495
  DataGridView Control ...................................................................................... 496
  DataGridView Sample Program ....................................................................... 497
  DataGridView Demo ......................................................................................... 498
  Performing a Query ........................................................................................... 504
  Lab 13B ............................................................................................................. 507
  Summary .......................................................................................................... 508

Chapter 14 Newer Features in Visual Basic....................................................... 515
  Local Type Inference ......................................................................................... 517
  Local Type Inference – Example ...................................................................... 518
  Object Initializers ............................................................................................. 519
  Array Initializers .............................................................................................. 520
  Anonymous Types ........................................................................................... 521
  Partial Methods ................................................................................................ 522
  Partial Method Definition ............................................................................... 523
  Partial Method Implementation ....................................................................... 524
  Test Program .................................................................................................... 525
  Running the Example ...................................................................................... 526
  Extension Methods .......................................................................................... 527
  Extension Methods Example ........................................................................... 528
  Collection Initializers ...................................................................................... 529
  Variance in Generic Interfaces ....................................................................... 530
  Covariance Example ......................................................................................... 531
  Variance with IComparer(Of T) ....................................................................... 532
  Interfaces with Variance Support .................................................................... 533
  Contravariance Example .................................................................................. 534
Chapter 1

.NET: What You Need to Know
.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 2012 as an effective environment for creating Visual Basic programs.

• Use the .NET Framework SDK documentation.
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.
.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.
.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.
.NET Program Example

- See SimpleCalc in the Chap01 folder.

1. Enter the program in a text editor.

```vbnet
' 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 2012 | Visual Studio Tools | Developer Command Prompt for VS2012. Navigate to the Chap01\SimpleCalc folder.

>`vbc SimpleCalc.vb`

3. Run your IL program SimpleCalc.exe

>`SimpleCalc`

area = 100
Viewing the Assembly

- You can view the assembly using the *ILDASM* tool\(^1\).

> ildasm SimpleCalc.exe

---

\(^1\) You can change the font size from the View menu.
Viewing Intermediate Language

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

```
.method public static void Main() cil managed
{
  .entrypoint
  .custom instance void [mscorlib]System.SThreadAttribute::.ctor() = (
    // Code size 26 (0x1a)
    .maxstack 2
    .locals init (int32 v_0,
                   int32 v_1,
                   int32 v_2)
  IL_0000:  ldc.i4.5  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_000a:  ldloc.0
  IL_000b:  box      [mscorlib]System.Int32
  IL_0014:  call      void [mscorlib]System.Console::WriteLine(string, object)

  IL_0019:  ret
} // end of method SimpleCalc::Main
```
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
Visual Studio 2012

While it is possible to write Visual Basic programs using any text editor, and compile them with the command-line compiler, it is very tedious to program that way.

An IDE makes the process of writing software much easier.

− An IDE provides convenience items, such as a syntax-highlighting editor.

− An IDE reduces the tedium of keeping track of configurations, environment settings and file organizations.

You may use Visual Studio 2012 throughout this course to create and compile your VB programs.

Visual Studio 2012 is discussed in more detail in Appendix A.
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. Leave .NET Framework 4.5 as the target framework. Leave checked “Create directory for solution”.

4. In the Name field, type SimpleCalcVs and for Location browse to C:\OIC\IntroVb\Demos. Click OK.

2 Examples in later chapters frequently do not have a directory for solution.
Visual Studio Solutions

• In Visual Studio 2012, 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.

![Solution Explorer](image-url)
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.
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.
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.
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).³

• 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!

³ 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.
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 2012 project in Chap01\SimpleCalcVs.

- Remember that you can always make the console window stay open by running without the debugger via Control + F5.
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

![Rectangle Calculator](image)

- We will discuss GUI programming using Visual Basic beginning in Chapter 10.
.NET Documentation

- MSDN documentation for the .NET Framework is included with Visual Studio 2012.
  - Use the menu Help | View Help. Other menu choices let you add and remove content and to set a preference for launching in Browser or Help Viewer.
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 2012 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.
Chapter 2

Fundamentals of Visual Basic Programming
Fundamentals of Visual Basic Programming

Objectives

After completing this unit you will be able to:

- Build and run Visual Basic programs.
- Describe the basic structure of Visual Basic programs.
- Describe the use of namespaces in Visual Basic.
- Describe data types available in Visual Basic.
- Perform calculations in Visual Basic.
- Use console I/O in Visual Basic console programs.
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.

- **In order 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 2012, Visual Basic .NET, and VB.NET are all synonymous.
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 Chap02\Hello\Hello\Hello.vb.

```vbnet
' Hello.vb

Module Hello

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

End Module
```
Compiling, Running (Command Line)

- The Visual Studio 2012 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
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.
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.
  – Beginning 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.
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\Hello2.vb.

' Hello2.vb

Imports System

Module Hello

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

End Module
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.

```vbnet
' Hello3.vb
Module Hello
    Sub Main()
        Console.WriteLine("Hello, world")
    End Sub
End Module
```
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.
Naming Standards

- **Visual Basic programmers must name many elements in their programs, including:**
  - variables
  - procedures
  - modules
  - classes

- **The rules govern the naming of these elements are:**
  - You may use only letters, digits, and the underscore.
  - You must begin with either a letter or an underscore.
  - You must have at least one letter in the name.

- **Visual Basic names are not case sensitive.**
  - The source code editor will convert all instances of the name to the case that was used when the identifier was defined.

- **However, the .NET CLR uses case-sensitive binding.**
  - This means other .NET languages that interact with your assembly must use the same case that you used when you defined the element.

Examples:

grossPay _dateOfCompletion
Sum CalculateArea
X1 Find_Last_Known_Address
Keywords

• **Visual Basic uses keywords to identify certain structural elements of a program.**

• **Keywords include:**

<table>
<thead>
<tr>
<th>And</th>
<th>ByRef</th>
<th>ByVal</th>
<th>Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catch</td>
<td>Class</td>
<td>Dim</td>
<td>Else</td>
</tr>
<tr>
<td>Error</td>
<td>For</td>
<td>If</td>
<td>Inherits</td>
</tr>
<tr>
<td>Me</td>
<td>Module</td>
<td>New</td>
<td>Optional</td>
</tr>
<tr>
<td>Step</td>
<td>Unicode</td>
<td>Throw</td>
<td>To</td>
</tr>
<tr>
<td>Try</td>
<td>WithEvents</td>
<td>While</td>
<td>Xor</td>
</tr>
</tbody>
</table>

• **There are over 150 keywords in Visual Basic .NET, and the complete list can be found in the Visual Basic Language Reference in the MSDN documentation.**

  – Visual Basic allows you to use keywords as identifiers if you surround every reference to them with square brackets:

Dim [Single] As Boolean  
Dim [Date] As String

  – However, with a little work, you can probably think of other identifiers which convey the same meaning without using keywords:

Dim Unmarried As Boolean  
Dim DateName As String
Multiple-File Program Structure

• **Visual Basic programs consist of a collection of modules.**
  
  – The functionality in each module should be related in some way.
  
  – Each module in the application is named and resides in a file with a .vb extension.

• **Console applications must have a subroutine called Main.**
  
  – This defines the *entry point* of the program, or the point at which execution begins.

• **See the example program** *UsingModules.*

```vbnet
' DisplayModule.vb
' Collection of related procedures

Module DisplayModule
    Public Sub SayHello()
        Console.WriteLine("Hello!")
    End Sub

    Public Sub RambleOn()
        Console.WriteLine("How now brown cow...")
    End Sub

    Public Sub SayGoodbye()
        Console.WriteLine("Goodbye!")
    End Sub
End Module
```
Using Procedures

• Any code in the application can call the Public procedures defined in DisplayModule.

' UsingModules.vb
' Example of using procedures in another module

Module UsingModules

    Sub Main()
        SayHello()
        RambleOn()
        SayGoodbye()
    End Sub

End Module
### Data Types

- The simple data types include:

<table>
<thead>
<tr>
<th>Visual Basic TYPE</th>
<th>.NET FRAMEWORK TYPE</th>
<th>MEMORY REQUIREMENTS</th>
<th>RANGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>System.Object</td>
<td>4 bytes</td>
<td>Any type can be stored in a variable of type Object</td>
</tr>
<tr>
<td>Byte</td>
<td>System.Byte</td>
<td>1 byte</td>
<td>0 to 255 (unsigned)</td>
</tr>
<tr>
<td>Char</td>
<td>System.Char</td>
<td>2 bytes</td>
<td>0 to 65535 (unsigned)</td>
</tr>
<tr>
<td>Short</td>
<td>System.Int16</td>
<td>2 bytes</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>Integer</td>
<td>System.Int32</td>
<td>4 bytes</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>Single</td>
<td>System.Single</td>
<td>4 bytes</td>
<td>single-precision floating point corresponding to the IEEE 754 standard</td>
</tr>
<tr>
<td>Double</td>
<td>System.Double</td>
<td>8 bytes</td>
<td>double-precision floating point corresponding to the IEEE 754 standard</td>
</tr>
<tr>
<td>Boolean</td>
<td>System.Boolean</td>
<td>4 bytes</td>
<td>True or False</td>
</tr>
<tr>
<td>String</td>
<td>System.String</td>
<td>10 bytes + (2 * string length)</td>
<td>0 to almost 2.1 billion Unicode characters</td>
</tr>
<tr>
<td>Decimal</td>
<td>System.Decimal</td>
<td>12 bytes</td>
<td>96-bit signed integers scaled by a variable power of 10; from 0 to 28 digits to the right of the decimal point</td>
</tr>
<tr>
<td>Date</td>
<td>System.DateTime</td>
<td>8 bytes</td>
<td>January 1, 1 CE (year 1) to December 31, 9999 and times from 00:00:00 to 23:59:59</td>
</tr>
</tbody>
</table>
IEEE 754 Floating Point Standard

- Microsoft has adopted the IEEE 754 floating-point standard for all .NET languages.

- The standard specifies that certain operations, which previously might have produced errors, now create special values.
  - **Not-a-Number** (also known as NaN) values result from dividing 0.0 by 0.0.
  - **Positive infinity** values result from dividing a non-zero positive value by 0.0.
  - **Negative infinity** values result from dividing a non-zero negative value by 0.0.

- See the example program *IEEE754*. 
IEEE 754 (Cont'd)

' IEEE754.vb
' Example of IEEE 754 special values

Module IEEE754
    Sub Main()
        Dim top As Single
        Dim bottom As Single

        top = 0
        bottom = 0
        Console.WriteLine(top / bottom)

        top = 18
        bottom = 0
        Console.WriteLine(top / bottom)
        top = -18
        bottom = 0
        Console.WriteLine(top / bottom)
    End Sub
End Module

• The output of the program looks like this:
  NaN
  Infinity
  -Infinity
Data Type Ranges

• Most of the numeric classes (Byte, Integer, Single, etc.) have a collection of methods and properties that can be very useful in daily programming:
  
  − Properties **MinValue** and **MaxValue** can be used to determine the minimum or maximum value of a specific data type.
  
  − See the example program **DataRanges**.

```csharp
Console.WriteLine("Integer Range:")
Console.WriteLine(Integer.MinValue & " to " & _
Integer.MaxValue)

Console.WriteLine("Single Range:")
Console.WriteLine(Single.MinValue & " to " & _
Single.MaxValue)

Console.WriteLine("Date Range:")
Console.WriteLine(Date.MinValue & " to " & _
Date.MaxValue)

output

Integer Range:
-2147483648 to 2147483647
Single Range:
-3.402823E+38 to 3.402823E+38
Date Range:
1/1/0001 12:00:00 AM to 12/31/9999 11:59:59 PM
```
• **A literal** is a value that is explicitly written in code.

```vbnet
a = 13    ' 13 is a literal
name = "Natalie"   ' "Natalie" is a literal
```

• **Visual Basic assumes that literals are typed.**
  
  − The type is determined based on the way the value is expressed.
  
  − If a literal is expressed without any suffix, its data type is either Integer, Double, String, or Boolean.

```vbnet
12            ' is an Integer literal
9.2           ' is a Double literal
"Hi"          ' is a String literal
False         ' is a Boolean literal
```

• **To explicitly specify the data type of a literal, you must use a suffix to indicate its type.**

```vbnet
12S           ' is a Short literal
12I           ' is an Integer literal
12L           ' is a Long literal
12F           ' is a Single literal
12R           ' is a Double literal
12D           ' is a Decimal literal
"L"C          ' is a Char literal
```
• Visual Basic assumes that integer-type literals are decimal (base 10).
  
  - You can specify that literals values are octal (base 8) or hexadecimal (base 16) by prefacing the literal with an ampersand (&) and the letter O (for octal) or H (for hexadecimal).

    `70         ' a decimal literal
    &O70       ' an octal literal - which equals
                ' the decimal value 56
    &H70       ' a hexadecimal literal - which
                ' equals the decimal value 112

• Date literals require a special character (#) to mark both the beginning and the ending of the literal string.
  
  - This string may include a date, a time, or a date and a time.
  
  - If the date is omitted, 1/1/0001 is assumed.
  
  - A time value may be specified using either a 24-hour or a 12-hour value. If the time omits the AM or PM, it is assumed to be a 24-hour time.
  
  - If the time is omitted, 12:00:00 AM is assumed.

    #9/5/1958#
    #9/5/1958 10:02:00 AM#
    #10:02:00 AM#
Variables

- **Variables** are symbolic names that are used to represent data in a program.
  - The variable `id` could hold an employee ID number and the variable `pay` could hold an employee’s pay rate.

- Variables are typically defined inside a procedure using the *Dim* statement.
  - The variables are local to the procedure and can be referenced only inside that procedure.
  - These variables are created when the procedure is invoked, and destroyed when it terminates.

```vba
Sub ProcessEmployeePayroll()
    Dim id As Integer
    Dim name As String
    Dim payRate As Decimal
    ...
End Sub
```

- Several variables (of the same type) may be declared in one *Dim* statement.

```vba
Dim grossPay, taxes, netPay As Single
Dim isValid, isWithinRange As Boolean
```

- Variables can also be defined within a module using the *Public* or *Private* statement.
  - These variables are created when the application is begins, and destroyed when the application terminates.
Variables (Cont'd)

- The variables are either local to the module (if declared as **Private**) or available throughout the application (if declared **Public**).

```vbnet
Module IOFunctions
    Private fileName As String
    ...
End Module

Module EmployeeFunctions
    Public id As Integer
    Public name As String
    Public payRate As Decimal
    ...
End Module
```

- **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 Compile properties.
    - We strongly urge you to keep **Option Explicit** set to **On** !!!
Initialization of Variables

- **Visual Basic automatically initializes** variables when they are declared.
  - All numeric variables are initialized to zero.
  - Boolean variables are initialized to False.
  - Dates are initialized to #1/1/0001 12:00:00 AM#

- **Variables may be explicitly initialized when they are declared.**
  - If a variable is initialized when it is declared, one variable may be declared per statement.

  Dim unitPrice As Decimal = 12.97
  Dim x, y As Integer = 100  ' generates an error
Type Checking

- Most compiled programming languages are typed.
  - That is, they check the data types of variables when they are used to make sure that the operation is allowed.
  - Visual Basic has a compiler option (*Option Strict*) that can indicate the level of type-checking that should be used.

- When *Option Strict* is set to *Off*, Visual Basic is very flexible about the operations it allows and performs *implicit conversions* for you.
Weak Type Checking Example

- See example program *WeakTypeChecking*.

' Example of Weak Type Checking
' (Option Strict off)

Module WeakTypeChecking
    Sub Main()
        Dim someString As String
        Dim someNumber As Integer
        someString = "7"

        ' string-to-integer assignment
        someNumber = someString
        Console.WriteLine("{0}", someString)
        Console.WriteLine("{0}", someNumber)
    End Sub
End Module

- When *Option Strict* is set to *On*, Visual Basic will not perform most implicit conversions for you and will generate errors instead.

  - When you compile the previous program, you receive the error

    "Option Strict disallows implicit conversions from String to Integer"

  on the line:

    someNumber = someString
Allowed Conversions

- If Option Strict is On, the compiler will not necessarily generate error messages on all data type mismatches.
  
  - For example, on numeric assignments, Visual Basic is primarily concerned about loss of precision.

```vbnet
Dim someFloat As Single
Dim anotherNumber As Integer

' if no loss of precision occurs, "obvious"
' implicit conversions are allowed and the
' following line is legal

someFloat = 4

' if loss of precision occurs, all implicit
' conversions are disallowed and the following line
' generates a syntax error

anotherNumber = someFloat
```
Constants

• When writing code, you are sometimes faced with a decision regarding values that don't change.

• For example, assume the local sales tax rate is 8.25 percent -- how do you represent it?
  – You might choose to represent it as a literal:

    tax = price * .0825

  – You might use a variable to represent the sales tax rate:

    Dim salesTaxRate As Single = .0825
    ...
    tax = price * salesTaxRate

  – The most desirable one approach, however, is to use a constant.

    Const salesTaxRate As Single = .0825
    ...
    tax = price * salesTaxRate

• A constant is similar to a variable; however, the compiler generates an error if an attempt is made to change its value.

    Const salesTaxRate As Single = .0825
    ...
    salesTaxRate = .0850      'error
Data Conversions

• Often, you need to use a value of one data type when a different data type is expected.
  – This will require that you convert the value expressed in one data type into another data type.
  – Visual Basic supports two types of conversions:

• An *implicit* conversion is performed silently by the compiler as needed.
  – A conversion is safe if there is no loss of precision (ex: conversions from *Integer* to *Long*).
  – A conversion might cause problems if there is a loss of precision (ex: conversions from *Single* to *Long*).

Dim n1 As Single
n1 = 46 ' implicitly convert from Integer to Single

• An *explicit* conversion is performed by the programmer using a conversion function.

Dim s As String = "43.99"
Dim n2 As Single = Convert.ToSingle(s) ' explicitly convert from String to Single
Data Conversions (Cont'd)

- There are two sets of functions that explicitly convert the data:

<table>
<thead>
<tr>
<th>Conversion to…</th>
<th>Visual Basic Conversion Function</th>
<th>.NET System(Convert Method)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>CByte</td>
<td>ToByte</td>
</tr>
<tr>
<td>Short</td>
<td>CShort</td>
<td>ToInt16</td>
</tr>
<tr>
<td>Integer</td>
<td>CInt</td>
<td>ToInt32</td>
</tr>
<tr>
<td>Long</td>
<td>CLng</td>
<td>ToInt64</td>
</tr>
<tr>
<td>Single</td>
<td>CSng</td>
<td>ToSingle</td>
</tr>
<tr>
<td>Double</td>
<td>CDbl</td>
<td>ToDouble</td>
</tr>
<tr>
<td>Decimal</td>
<td>CDec</td>
<td>ToDecimal</td>
</tr>
<tr>
<td>Char</td>
<td>CChar</td>
<td>ToChar</td>
</tr>
<tr>
<td>String</td>
<td>CStr</td>
<td>ToString</td>
</tr>
<tr>
<td>Date</td>
<td>CDate</td>
<td>ToDateTime</td>
</tr>
<tr>
<td>Boolean</td>
<td>CBool</td>
<td>ToBoolean</td>
</tr>
</tbody>
</table>

- With *Option Explicit* turned *On*, you will need to explicitly convert many values in your application.

```vbnet
Dim pi As Single = 3.14 ' generates a compiler error because 3.14 is a double
Dim pi As Single = CSng(3.14)
Dim pi As Single = Convert.ToSingle(3.14)
```
Operators and Expressions

• Now that we can declare variables, we need to be able to build complex expressions

• Visual Basic has a set of operators that have been enhanced since VB6.
  – Unary operators take a single operand. A simple example is unary minus, which changes the sign of a number.
  – Binary operators are the most common and take two operands. Examples include simple arithmetic operators such as addition and subtraction.

• For illustrations of the use of a number of operators, see the example program OperatorSummary.
Arithmetic Operators

• The arithmetic operators may be applied to any of the numerical data types of integer, floating-point, and decimal.

• The evaluation of any expression will always result in a value that represents the most precise operand.
  
  − For example, if both operands of a multiply are Integer, the result will be Integer.

  − However, if one operand is an Integer and the other operand is a Double, the result will be a Double because Double is a higher level of precision.

• Unary operators are:
  
  − negate a value
  + indicate a positive value

  Dim a, b As Integer
  a = +b
  a = -b

• Binary operators are:

  + addition
  − subtraction
  * multiplication
  / floating-point division
  \ integer division
  Mod remainder from integer division
  ** exponenetiation
Arithmetic Operators (Cont'd)

2.5 + 1    ' result is 2.5
2.5 - 1    ' result is 1.5
2.5 * 3     ' result is 7.5
2.5 / 1.3    ' result is 1.92307692307692
2.5 \ 1.3   ' result (rounded) is 2
5 / 3     ' result is 1.66666666666667
5 \ 3     ' result is 1
5 Mod 3     ' result is 2
2.5 ^ 1.3    ' result is 3.2909551083559

• You must be careful with certain mathematical operations because the result might be so large that it cannot be represented.
  – It will generate an **OverflowException** runtime error.
  – This error can be trapped.

```vbnet
Dim n1 As Integer = Integer.MaxValue
Dim n2 As Integer
Try
    n2 = n1 * 2
    Console.WriteLine("Answer = " & n2)
Catch e As OverflowException
    Console.WriteLine("
    "Operation resulted in an overflow!"
)
End Try
```

• Exceptions are discussed in Chapter 6.
String Operators

- Visual Basic allows you to use both the addition operator (+) and the concatenation operator (&) to perform string concatenation.

Dim s1 As String = "Hello"
Dim s2 As String = "World"
Dim s3 As String
s3 = s1 & " " & s2 ' result is Hello World

- But it is a good idea to form the habit of using the concatenation operator.
  
  - In some circumstances errors can arise from the use of the addition operator, in situations where there could be confusion with numerical addition.
Relational Operators

- Visual Basic has six operators for making comparisons between expressions.

  =  equality

  <>  inequality

  <  less than

  <=  less than or equal to

  >  greater than

  >=  greater than or equal to

- These operators are most often used when comparing two values to decide what course of action should be taken.

  If a < b Then
      ...
  End If

  If name = "Brenda" Then
      ...
  End If
Logical Operators

- You can build complex logical expressions using the logical operators:
  
  And
  Or
  Xor
  Not

  If num < minValue Or num > maxValue Then
    Console.WriteLine("Error: number out of range")
  End If

  Dim isDone As Boolean = False
  Do While (Not isDone)
    ...
  End While

- Visual Basic provides two logical operators to help evaluate expressions as efficiently as possible.
  - These operators short circuit the evaluation process
  - That is, they cause the evaluation process to stop examining the expression when the result of evaluation is known

  AndAlso

  OrElse
Logical Operators (Cont’d)

− In the following If, the AndAlso operator is used to prohibit the evaluation of the second part of the expression when the value of age is NOT greater than 18.

If age > 18 AndAlso state = "TX" Then
  ...
End If

− In the following If, the OrElse operator is used to prohibit the evaluation of the second part of the expression when the value of inputVal is less than 0.

If inputVal < 0 OrElse inputVal > 100 Then
  ...
End If
Bitwise Operators

• Visual Basic supports the following operators that manipulate operands at a bit level

And       performs bitwise and
Or        performs bitwise or
Xor       performs bitwise exclusive or
Not       performs one compliment operation

Dim b1, b2, b3 As Byte
b1 = 19    ' Bit pattern of b1 is 00010011
b2 = 7     ' Bit pattern of b2 is 00000111

b3 = b1 And b2    ' Bit pattern is 00000011 (or 3)
b3 = b1 Or b2     ' Bit pattern is 00010111 (or 23)
b3 = b1 Xor b2    ' Bit pattern is 00010100 (or 20)
b3 = Not b1      ' Bit pattern is 11101100 (or 236)
Assignment Operators

- The assignment operator (=) is used to assign a value to a variable.

\[ x = 5 + 3 \]

- Visual Basic also has compound assignment operators that perform operations as part of the assignment.

\[ x += 2 \quad \text{`equivalent to} \quad x = x + 2 \]
\[ x += 2 * b \quad \text{`equivalent to} \quad x = x + (2 * b) \]

- The compound assignment operators are:

  * `*=` 'multiply and assign'
  /=` 'divide and assign'
  `+=` 'add and assign'
  `-=` 'subtract and assign'
  `\=` 'divide and assign'
  `&=` 'concatenate and assign'
## Operator Precedence

- The precedence table for operators is shown below.
  - It includes a few operators we have not discussed yet.

<table>
<thead>
<tr>
<th>Category</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponentiation</td>
<td>^</td>
</tr>
<tr>
<td>Negation</td>
<td>-</td>
</tr>
<tr>
<td>Multiplication and division</td>
<td>* /</td>
</tr>
<tr>
<td>Integer division</td>
<td>\</td>
</tr>
<tr>
<td>Modulo (remainder) arithmetic</td>
<td>Mod</td>
</tr>
<tr>
<td>Addition, subtraction, string</td>
<td>+ - + (string)</td>
</tr>
<tr>
<td>concatenation</td>
<td></td>
</tr>
<tr>
<td>String concatenation</td>
<td>&amp;</td>
</tr>
<tr>
<td>Comparison</td>
<td>= &lt;&gt; &lt;= &gt; &gt;= Like Is TypeOf..Is</td>
</tr>
<tr>
<td>Negation</td>
<td>Not</td>
</tr>
<tr>
<td>Logical/Bitwise conjunction</td>
<td>And AndAlso</td>
</tr>
<tr>
<td>Logical/Bitwise disjunction</td>
<td>Or OrElse Xor</td>
</tr>
</tbody>
</table>

- A simple example of precedence is addition and multiplication.

```vba
x = 5 * 8 + 13  ' multiply first so result is 53
x = 5 * (8 + 13) ' add first so result is 105
```
Console I/O

- Console IO is provided in the *Console* class in the *System* namespace.

- Although most of your professional applications will be built using Windows Forms or ASP.NET, there is value in studying console IO.
  - It provides an easy way to learn the language without complicating the application with GUI issues.
  - The output formatting techniques can be applied to string output in both Windows Forms and ASP.NET.

- Console I/O is illustrated in the example program *ConsoleIO*. 
Console Input

- The `Console` class provides a `ReadLine` method to read a line of data from the standard input device.
  - The user must press the Enter key before the input is sent to your program.
  - `ReadLine` returns the string that the user typed, up to but not including the carriage return / line feed.
  - Because `ReadLine` always returns a string, you must use the conversion functions discussed earlier to convert the data that is entered to the proper data type.

```vbnet
Dim inputBuffer As String
Dim payRate As Decimal
Dim hrsWorked As Single

Console.WriteLine("Enter Pay Rate: ")
inputBuffer = Console.ReadLine()
payRate = Convert.ToDecimal(inputBuffer)

Console.WriteLine("Enter Hours Worked: ")
inputBuffer = Console.ReadLine()
hrsWorked = Convert.ToSingle(inputBuffer)
```
Console Output

• The *Console* class supports two simple methods for performing output:
  
  – **WriteLine** writes a string followed by a new line character.
  
  – **Write** writes just the string without the new line character.

```vbnet
Dim x As Integer = 24
Dim y As Integer = 5
Dim z As Integer = x * y
Console.Write(Convert.ToString(x) & " times " & Convert.ToString(y))
Console.WriteLine(" is " + Convert.ToString(z))
```

• The output of this example is as follows:

```plaintext
24 times 5 is 120
```
Placeholders

- A more convenient way to build an output string is to use placeholders,
  - `{0}`, `{1}`, and so on are placed in an output string to indicate where actual data should appear.

  Console.WriteLine("{0} times {1} is {2}", x, y, z)

- Placeholders may contain three types of formatting information:
  - Width specification `{n,w}`, where `n` is the placeholder number and `w` is the width (positive for right justification and negative for left justification).
  - If the width specified is not large enough for the value, it will be ignored.
  - Format specification `{n:S}`, where `n` is the placeholder number and `S` is a format string.
  - Width and format specification `{n,w:S}`, where `n` is the placeholder number, `w` is the width, and `S` is a format string.
Placeholders (Cont'd)

- **The format characters include:**

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Decimal integer</td>
<td>Can only be used with integer values</td>
</tr>
<tr>
<td>E</td>
<td>Exponential notation</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Fixed-point</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>General (E or F)</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>Number with embedded commas</td>
<td>Can only be used with integer values</td>
</tr>
<tr>
<td>X</td>
<td>Hexadecimal</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Currency (locale specific)</td>
<td></td>
</tr>
</tbody>
</table>

- **The width-only specification can be used to build tabular output.**

```csharp
Console.WriteLine("{0,-10} {1,3}", name, age)
```

- If the above line were placed in a loop, the output might resemble this:

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>103</td>
</tr>
<tr>
<td>Ralph</td>
<td>32</td>
</tr>
<tr>
<td>Jo</td>
<td>24</td>
</tr>
<tr>
<td>Jessie</td>
<td>1</td>
</tr>
</tbody>
</table>

- **The format specification is used to make numeric output more readable.**

- It is very useful to ensure that values are displayed with the desired number of digits after the decimal place.
Dim productName As String = "BBQ Ribs (1 lb)"
Dim price As Single = 8.5

Console.WriteLine("{0} is priced at $" _
 & "{1,8:F2}", productName, price)
Console.WriteLine("{0} is priced at $" _
 & "{1,8:F2}", productName, price)
Console.WriteLine("{0} is priced at "$ _
 & "{1,8:C}", productName, _
Convert.ToDecimal(price))

- The output of the program above would look like this:

BBQ Ribs (1 lb) is priced at $8.50
BBQ Ribs (1 lb) is priced at $  8.50
BBQ Ribs (1 lb) is priced at $8.50
A Practical Example

To demonstrate the Visual Basic skills we have acquired, we will examine the SavingsCalculator program.

- It computes accumulations in an Individual Retirement Account with annual deposits and compound interest.
- The program will accept the following inputs:

Annual Deposit = A
Interest Rate = R
Number of Years = N

- The following formula assumes that a deposit is made at the end of each year and that interest is compounded annually:

\[
\text{Total Accumulation} = A \times \frac{(1 + R)^N - 1}{R}
\]

Imports System
Module SavingsCalculator
Sub Main()
    Dim depositAmount As Decimal
    Dim interestRate As Single
    Dim numYears As Integer

    Console.Write("Enter amount of deposit: ")
    depositAmount = Convert.ToDecimal(Console.ReadLine())
    ...
End Sub

A Practical Example (Cont'd)

```vbnet
Console.Write("Enter interest rate: ")
interestRate = _
    Convert.ToSingle(Console.ReadLine())

Console.Write("Enter number of years: ")
numYears = _
    Convert.ToInt32(Console.ReadLine())

Dim endingValue As Decimal
endingValue = depositAmount * _
    ((1 + interestRate) ^ numYears) - 1) _
    / interestRate

Console.WriteLine("Ending Value = {0:C1}", _
    endingValue)
End Sub

End Module
```

- The following shows the output of one run:

Enter amount of deposit: 1000
Enter interest rate: .07
Enter number of years: 20
Ending Value = $40995.49
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.
Lab 2

New Car Loan Calculator

In this lab, you will build a program that calculates the monthly payment of a car loan.

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

Suggested time: 45 minutes
Summary

- Visual Basic is a high-level, object-oriented programming language with a variety of data types and control structures.

- Visual Basic supports a rich collection of data types that include Integer, Long, Single, Double, Date, and String.

- As with other programming languages, you can build variables and constants in Visual Basic.

- Visual Basic has a rich collection of operators that can be used to build complex expressions.

- The `System.Console` class includes methods console I/O, such as `ReadLine` and `WriteLine`.

- The `System.Console` class has extensive formatting capabilities controlled through placeholders.
Lab 2

New Car Loan Calculator

Introduction

In this lab, you will build a program that calculates the monthly payment of a new car loan. The user will be prompted for the car's purchase price, the down payment, the interest rate, and length (in months) of the loan. The program will calculate the sales tax, title fee, total purchase price of the purchase as well as the monthly payment of the new car loan. (We will return to this car loan example in Chapters 6 and 10.)

Suggested Time: 45 minutes

Root Directory: OIC\IntroVb

Work Directory: Labs\Lab2
Solutions Directory: Chap02\CarLoan\Version1

Instructions

1. Create a new Visual Basic Console Application and name it CarLoan. Change the Module1.vb file name to CarLoan.vb and notice that the module name is automatically changed to CarLoan, and the project will start there.

2. Declare Visual Basic variables for the variables shown in the formula in Step 4 below.

3. Prompt the user for the car's purchase price, the down payment, the interest rate, and length (in months) of the loan.

4. Use the following formulas to calculate the monthly payment,

   where:  
   
   **P** is the purchase price
   **T** is the sales tax
   **TL** is the title fee
   **D** is the down payment
   **FA** is the finance amount
   **R** is the interest rate (ex: 6.75)
   **N** is the number of months of the loan (ex: 24 or 60)
   **Payment** is monthly payment

   \[
   T = P \times 0.0625
   \]

   \[
   TL = 25
   \]
\[ FA = P + T + TL - D \]

\[ Payment = \frac{FA \times (R/1200)}{1 - (1 + R/1200)^{-Y}} \]

5. Display the payment.

6. Build and test your application.
Chapter 13

Database Programming
Database Programming

Objectives

After completing this unit you will be able to:

- Understand the key concepts in the ADO.NET data access programming model.
- Use the ADO.NET classes to build applications that retrieve and update data from a data source.
- Use Visual Studio 2012 to efficiently implement database programs.
- Use the data binding facilities in Windows Forms and ADO.NET.
ADO.NET

• The .NET Framework has introduced a new set of database classes designed for loosely coupled, distributed architectures.
  – These classes are referred to as ADO.NET.

• ADO.NET uses the same access mechanisms for local, client-server, and Internet database access.
  – It can be used to examine data as relational data or as hierarchical (XML) data.

• ADO.NET can pass data to any component using XML and does not require a continuous connection to the database.

• A more traditional connected access model is also available.
ADO.NET Architecture

• The *DataSet* class is the central component of the disconnected architecture.
  
  − A dataset can be populated from either a database or from an XML stream.
  
  − From the perspective of the user of the dataset, the original source of the data is immaterial.
  
  − A consistent programming model is used for all application interaction with the dataset.

• The second key component of ADO.NET architecture is the *.NET Data Provider*, which provides access to a database, and can be used to populate a dataset.
  
  − A data provider can also be used directly by an application to support a connected mode of database access.
• The figure illustrates the overall architecture of ADO.NET.
.NET Data Providers

• A .NET data provider is used for connecting to a database.
  – It provides classes that can be used to execute commands and to retrieve results.
  – The results are either used directly by the application, or else they are placed in a dataset.

• A .NET data provider implements four key interfaces:
  – IDbConnection is used to establish a connection to a specific data source.
  – IDbCommand is used to execute a command at a data source.
  – IDataReader provides an efficient way to read a stream of data from a data source. The data access provided by a data reader is forward-only and read-only.
  – IDbDataAdapter is used to populate a dataset from a data source.

• The ADO.NET architecture specifies these interfaces, and different implementations can be created to facilitate working with different data sources.
  – A .NET data provider is analogous to an OLE DB provider, but the two should not be confused. An OLE DB provider implements COM interfaces, and a .NET data provider implements .NET interfaces.
Programming with ADO.NET

Interfaces

- In order to make your programs more portable, you should endeavor to program with the interfaces rather than using specific classes directly.
  - In our example programs we will illustrate using interfaces to talk to an Access database (using the OleDb data provider) and a SQL Server database (using the SqlServer data provider).

- Classes of the OleDb provider have a prefix of OleDb, and classes of the SqlServer provider have a prefix of Sql.
  - The table shows a number of parallel classes between the two data providers and the corresponding interfaces.

<table>
<thead>
<tr>
<th>Interface</th>
<th>OleDb</th>
<th>SQL Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDbConnection</td>
<td>OleDbConnection</td>
<td>SqlConnection</td>
</tr>
<tr>
<td>IDbCommand</td>
<td>OleDbCommand</td>
<td>SqlCommand</td>
</tr>
<tr>
<td>IDataReader</td>
<td>OleDbDataReader</td>
<td>SqlDataReader</td>
</tr>
<tr>
<td>IDbDataAdapter</td>
<td>OleDbDataAdapter</td>
<td>SqlCommandAdapter</td>
</tr>
<tr>
<td>IDbTransaction</td>
<td>OleDbTransaction</td>
<td>SqlTransaction</td>
</tr>
<tr>
<td>IDataParameter</td>
<td>OleDbParameter</td>
<td>SqlParameter</td>
</tr>
</tbody>
</table>

- Classes such as DataSet which are independent of any data provider do not have any prefix.
.NET Namespaces

- ADO.NET classes are found in the following namespaces:
  - **System.Data** consists of classes that constitute most of the ADO.NET architecture.
  - **System.Data.OleDb** contains classes that provide database access using the OLE DB data provider.
  - **System.Data.SqlClient** contains classes that provide database access using the SQL Server data provider.
  - **System.Data.OracleClient** contains classes that provide database access using the Oracle data provider. Both Microsoft and Oracle supply data providers for Oracle.
  - **System.Data.SqlTypes** contains classes that represent data types used by SQL Server.
  - **System.Data.Common** contains classes that are shared by data providers.
Connected Data Access

• The connection class (*OleDbConnection* or *SqlConnection*) is used to manage the connection to the data source.
  
  – It has properties `ConnectionString`, `ConnectionTimeout`, and so forth.

  – There are methods for `Open`, `Close`, transaction management, etc.

• A *connection string* is used to identify the information the object needs to connect to the database.

  – You can specify the connection string when you construct the connection object, or by setting its properties.

  – A connection string contains a series of `argument = value` statements separated by semicolons.

• To program in a manner that is independent of the data source, you should obtain an interface reference of type `IDbConnection` after creating the connection object, and you should program against this interface reference.
Sample Database

- Our sample database, *SimpleBank*, stores account information for a small bank. Two tables:

1. **Account** stores information about bank accounts. Columns are **AccountId**, **Owner**, **AccountType** and **Balance**. The primary key is **AccountId**.

2. **BankTransaction** stores information about account transactions. Columns are **AccountId**, **XactType**, **Amount** and **ToAccountId**. There is a parent/child relationship between the **Account** and **BankTransaction** tables.

- There are SQL Server 2012 and Access versions of this database.
  - We will be using SQL Server 2012 LocalDB, which comes automatically with Visual Studio 2012.
  - There is also built-in support for Access as well, so no special DBMS software is required for this chapter.

- The database files are in the folder *OIC\Data*.
  - The SQL Server database file is **SimpleBank.mdf**
  - The Access database file is **SimpleBank.mdb**.
Example: Connecting to SQL Server

− See SqlConnectOnly.

Imports System.Data.SqlClient

Module Module1

Sub Main()
    Dim connStr As String = "Data Source=(LocalDB)\v11.0;AttachDbFilename=c:\OIC\Data\SimpleBank.mdf;Integrated Security=True"
    Dim conn As New SqlConnection()
    conn.ConnectionString = connStr
    Console.WriteLine("Using SQL Server to access SimpleBank")
    Console.WriteLine("Database state: " & conn.State.ToString())
    conn.Open()
    Console.WriteLine("Database state: " & conn.State.ToString())
End Sub
End Class

Output:

Using SQL Server to access SimpleBank
Database state: Closed
Database state: Open
ADO.NET Class Libraries

- To run a program that uses the ADO.NET classes, you must be sure to set references to the appropriate class libraries. The following libraries should usually be included:
  - System.dll
  - System.Data.dll
  - System.Xml.dll (needed when working with datasets)

- References to these libraries are set up automatically when you create a Windows Form or Console in Visual Studio 2012.
  - The figure shows the references in a Console project, as created by Visual Studio 2012.

<table>
<thead>
<tr>
<th>Reference Name</th>
<th>Type</th>
<th>Version</th>
<th>Copy Local</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>.NET</td>
<td>4.0.0.0</td>
<td>False</td>
<td>C:\Program Files\Reference Assemblies\Microsoft\Framework</td>
</tr>
<tr>
<td>System.Core</td>
<td>.NET</td>
<td>4.0.0.0</td>
<td>False</td>
<td>C:\Program Files\Reference Assemblies\Microsoft\Framework</td>
</tr>
<tr>
<td>System.Data</td>
<td>.NET</td>
<td>4.0.0.0</td>
<td>False</td>
<td>C:\Program Files\Reference Assemblies\Microsoft\Framework</td>
</tr>
<tr>
<td>System.Data.DatasetExtensions</td>
<td>.NET</td>
<td>4.0.0.0</td>
<td>False</td>
<td>C:\Program Files\Reference Assemblies\Microsoft\Framework</td>
</tr>
<tr>
<td>System.Deployment</td>
<td>.NET</td>
<td>4.0.0.0</td>
<td>False</td>
<td>C:\Program Files\Reference Assemblies\Microsoft\Framework</td>
</tr>
<tr>
<td>System.Xml</td>
<td>.NET</td>
<td>4.0.0.0</td>
<td>False</td>
<td>C:\Program Files\Reference Assemblies\Microsoft\Framework</td>
</tr>
<tr>
<td>System.Xml.Linq</td>
<td>.NET</td>
<td>4.0.0.0</td>
<td>False</td>
<td>C:\Program Files\Reference Assemblies\Microsoft\Framework</td>
</tr>
</tbody>
</table>
Connecting to an OLE DB Data Provider

- To connect to an OLE DB data provider instead, you need to change the namespace you are importing and instantiate an object of the `OleDbConnection` class.
  - You must provide a connection string appropriate to your OLE DB provider.
  - We are going to use the Jet OLE DB provider, which can be used for connecting to an Access database.
  - The program `JetConnectOnly` illustrates connecting to the Access database `SimpleBank.mdb`

```vbnet
Imports System.Data.OleDb

Module Module1

Sub Main()
    Dim connStr As String = "Provider=Microsoft.Jet.OLEDB.4.0;" & "Data Source=C:\OIC\Data\SimpleBank.mdb"
    Dim conn As New OleDbConnection()
    conn.ConnectionString = connStr
    Console.WriteLine("
    "Using Access DB SimpleBank.mdb")
    Console.WriteLine("
    "Database state: " & conn.State.ToString())
    conn.Open()
    Console.WriteLine("
    "Database state: " & conn.State.ToString())
End Sub

End Class
```
Importing a Namespace

• Visual Studio 2012 offers a nice trick to make it easier to import a needed namespace in your program.

• Place the cursor over the .NET Framework class that is not recognized and press Ctrl + . (“control dot”).
  – A popup gives us an option to import the namespace. Just press Enter.

```vbnet
Imports System.Data.OleDb
Module Module1
Sub Main()
    Dim connStr As String = "Provider=Microsoft.Jet.OLEDB.4.0;" & "Data Source=\Evaluation Copy\.
    Dim conn As New OleDbConnection()
    conn.ConnectionString = connStr
    Console.WriteLine("Using Access")
    Console.WriteLine("Database")
    conn.Open()
    Console.WriteLine("Press Enter")
    Console.ReadLine()
End Sub
End Module
```

• The necessary namespace will be imported.

Imports System.Data.OleDb
Module Module1
Sub Main()
    Dim connStr As String = ...
    Dim conn As New OleDbConnection()
Using Commands

- After we have opened a connection to a data source, we can create a command object, which will execute a query against a data source.
  - Depending on our data source, we will create either a `SqlCommand` object or an `OleDbCommand` object.
  - In either case, we will initialize an interface reference of type `IDbCommand`, which will be used in the rest of our code, again promoting relative independence from the data source.

- The table summarizes some of the principle properties and methods of `IDbCommand`.

<table>
<thead>
<tr>
<th>Property or Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CommandText</td>
<td>Text of command to run against the data source</td>
</tr>
<tr>
<td>CommandTimeout</td>
<td>Wait time before terminating command attempt</td>
</tr>
<tr>
<td>CommandType</td>
<td>How CommandText is interpreted (e.g. Text, StoredProcedure)</td>
</tr>
<tr>
<td>Connection</td>
<td>The IDbConnection used by the command</td>
</tr>
<tr>
<td>Parameters</td>
<td>The parameters collection</td>
</tr>
<tr>
<td>Cancel</td>
<td>Cancel the execution of an IDbCommand</td>
</tr>
<tr>
<td>ExecuteReader</td>
<td>Obtain an IDataReader for retrieving data (SELECT)</td>
</tr>
<tr>
<td>ExecuteNonQuery</td>
<td>Execute a SQL command such as INSERT, DELETE, etc.</td>
</tr>
</tbody>
</table>
Creating a Command Object

- The code fragments shown below are from the ConnectedSql program, which illustrates performing various database operations on the SimpleBank database.
  - For an Access version, see ConnectedJet.

- The following code illustrates creating a command object and returning an IDbCommand interface reference.

  ```vba
  Private Function CreateCommand(ByVal query As String) As IDbCommand
    Return New SqlCommand(query, sqlConn)
  End Function
  ```

- Note that we return an interface reference, not an object reference.
  - Using the generic interface IDbCommand makes the rest of our program independent of a particular database.
The following code illustrates executing a SQL DELETE statement using a command object.

- We create a query string for the command, and obtain a command object for this command.
- The call to `ExecuteNonQuery` returns the number of rows that were updated.

```vbnet
Private Sub RemoveAccount(ByVal id As Integer)
    Dim query As String = "delete from Account where AccountId = " & id
    Dim command As IDbCommand = CreateCommand(query)
    Dim numrow As Integer = command.ExecuteNonQuery()
    Console.WriteLine("{0} rows updated", numrow)
End Sub
```
Using a Data Reader

- After we have created a command object, we can call the `ExecuteReader` method to return an `IDataReader`.
  - With the data reader we can obtain a read-only, forward-only stream of data.
  - This method is suitable for reading large amounts of data, because only one row at a time is stored in memory.
  - When you are done with the data reader, you should explicitly close it. Any output parameters or return values of the command object will not be available until after the data reader has been closed.

- Data readers have an `Item` property that can be used for accessing the current record.
  - The `Item` property accepts either an integer (representing a column number) or a string (representing a column name).
  - The `Item` property is the default property and can be omitted if desired.

- The `Read` method is used to advance the data reader to the next row.
  - When it is created, a data reader is positioned before the first row.
  - You must call `Read` before accessing any data. `Read` returns true if there are more rows, and otherwise false.
Data Reader: Code Example

- The code illustrates using a data reader to display results of a SELECT query.
  - Sample program is still in ConnectedSql.

```vbnet
Private Sub ShowList()
    Dim query As String = "select * from Account"
    Dim command As IDbCommand = CreateCommand(query)
    Dim reader As IDataReader = command.ExecuteReader()
    Do While reader.Read()
        Console.WriteLine("{0,3}  {1,-10}  {2:C} {3}",
            reader("AccountId"), reader("Owner") _
            reader("Balance"), reader("AccountType"))
    Loop
    reader.Close()
End Sub
```
Lab 13A

Bank Database Program

In this lab, you will build a Windows Forms application that provides a graphical user interface to the Simple Bank database.

Detailed instructions are contained in the Lab 13A write-up at the end of the chapter.

Suggested time: 30 minutes
Disconnected Datasets

- A **DataSet** stores data in memory and provides a consistent relational programming model that is the same whatever the original source of the data.
  - Thus, a **DataSet** contains a collection of tables and relationships between tables.
  - Each table contains a primary key and collections of columns and constraints, which define the schema of the table, and a collection of rows, which make up the data stored in the table.
Data Adapters

• A data adapter provides a bridge between a disconnected dataset and its data source.
  – Each .NET data provider provides its own implementation of the interface `IDbDataAdapter`.
  – The OLE DB data provider has the class `OleDbDataAdapter`, and the SQL data provider has the class `SqlDataAdapter`.

• A data adapter has properties for `SelectCommand`, `InsertCommand`, `UpdateCommand`, and `DeleteCommand`.
  – These properties identify the SQL needed to retrieve data, add data, change data, or remove data from the data source.

• A data adapter has the `Fill` method to place data into a dataset. It has the `Update` command to update the data source using data from the dataset.
Data Bound Controls

- All of our demonstration programs so far have been console applications.
  - We have done this in order to focus on the coding of database functionality, without being distracted by issues concerning user interface.
  - Also, by catching exceptions in our command loop, we are able to conveniently display exception information, making it easy for you to experiment and observe the results, even for exceptional situations.

- Naturally, in practice you will want to create an attractive user interface, either as a Windows application or a Web application.

- In this section we will give a brief introduction to providing a graphical user interface to a database application using Windows Forms.

- There are two approaches that can be followed.
  - The first is to write specific code to populate controls with data that has been retrieved from a database. You did that in Lab 13A.
  - The second is to use data bound controls. We will illustrate this second approach by showing how to bind a dataset to a DataGridView control.
DataGridView Control

- The DataGridView control is new with .NET 2.0 and is the preferred control for interacting with tabular data.
- Binding data to a DataGridView control is simple, and is often simply a matter of setting the DataSource property.
- It will bind to any class that implements one of the following interfaces:
  - IList, including one-dimensional arrays and array lists
  - IListSource, including DataSet and DataTable
  - IBindingList, such as GenericBindingList
  - IBindingListView, such as BindingSource
- Most frequently, you will bind to a BindingSource component, which is in turn bound to a data source.
- Visual Studio 2012 provides a wizard interface to facilitate establishing bindings.
DataGridView Sample Program

• We’ll demonstrate using a DataGridView control in an application that accesses the SmallPub database.
  – You should have SQL Server 2012 LocalDB installed.
  – The database file SmallPub.mdf is located in the directory C:\OIC\Data.

• This database is for use by a small publisher of technical books. There are two tables:
  – Category maintains a list of categories for books.
  – Book is a list of the books published.

• The sample program shows a list of all books in a category that is selected from a dropdown list.
  – See ShowBooks\Step2.
DataGridview Demo

- We’ll create this application from scratch, using Visual Studio.

1. Create a new Windows Forms Application **ShowBooks**. Save the solution of the new project in the **Demos** directory.

2. Increase the width of the main form somewhat, comparable to what is shown in the screen capture in the preceding page.

3. Drag a DataGridView control from the Data group in the ToolBox onto your form. Note that the “smart tag” for this control is expanded, allowing you to set up a binding. Drop down Choose Data Source and click on Add Project Data Source.
DataGridView Demo (Cont’d)

4. Make sure that Database is specified for the source of the data. Click Next.

5. A screen comes up letting you choose the kind of database model to use.

![Data Source Configuration Wizard]

6. Choose Dataset.
DataGridView Demo (Cont’d)

7. In the next screen, to choose the connection, click the New Connection button.

8. In the Add Connection dialog, make sure that SQL Server Database File is selected as the data source, and browse to the SmallPub.mdf file. You may test the connection if you wish.

![Add Connection Dialog]

9. Click OK. Back in the wizard, click Next.

10. You will be asked if you want to copy the database file to your project. Say No.
DataGridView Demo (Cont’d)

11. You will be asked if you want to save the connection string to the application configuration file. Leave the checkbox at Yes, and click Next.

12. In “Choose your database objects” expand Tables and check both the Book and Category tables.

13. Click Finish.

14. You will now have an opportunity to select which table in the DataSet you want to bind to the DataGridView control. (If the smart tag has closed, open it up again to finish binding the control to the project data source that you have just created.)
15. If necessary, open up the Project Data Sources to show the SmallPubDataSet and its tables.

![DataGridView Tasks]

16. Click on the link to the Book table.

17. Visual Studio will now create several components for you: a DataSet, a BindingSource, and a TableAdapter. See the screen capture on the following page. (At some point you will also want to resize the DataGridView control to provide adequate width for displaying all the columns.)
18. You will now see column headings displayed in the grid.
    Build and run. You will see all the books displayed.
Performing a Query

- We would now like to provide a facility for the user to show only the books in a particular category.

19. Open up the smart tag again, and click on the Add Query link.

20. Name the query “BooksByCat” and specify a WHERE clause, making the entire query:

   ```sql
   SELECT BookId, Title, CategoryId, PubYear
   FROM dbo.Book
   WHERE CategoryId = @CategoryId
   ```

---

1 I encountered an apparent bug in Visual Studio, not seeing the bottom two links at first. They appeared after I tried some of the other links.
Performing a Query (Cont’d)

21. When you are satisfied with your query, click OK.
Performing a Query (Cont’d)

22. A ToolStrip is added to your form to run the query. Build and run your application. Try entering a particular CategoryId, and then click the BooksByCat button.

![Show Books by Category](image)

23. You have a simple database query application, and you have not written a line of code!

24. Make any desired tweaks to the UI, such as making the DataGridView control larger. Change the title of your form to “Show Books by Category.” You are now at Step 1.

25. Examine Step 2, which replaces the automatically generated ToolStrip interface with your own combobox for displaying the available categories.
Lab 13B

Data Binding

In this lab, you will enhance the data binding demo program to specify the CategoryId by means of a dropdown combobox, which is bound to the Category table of the DataSet. You can use the code generated by Visual Studio for the Tool Strip as a guide for your own code. When done, you can delete the Tool Strip.

Detailed instructions are contained in the Lab 13B write-up at the end of the chapter.

Suggested time: 30 minutes
Summary

- ADO.NET provides a set of classes that can be used to interact with data providers.
- You can access data sources in either a connected or disconnected mode.
- The DataReader can be used to build interact with a data source in a connected mode.
- The DataSet can be used to interact with data from a data source without maintaining a constant connection to the data source.
- The DataSet can be populated from a database using a DataAdapter.
- You can use Visual Studio 2012 to efficiently implement database programs.
- You can easily display data from a dataset on a form by binding the dataset to a DataGridView control.
Lab 13A

Bank Database Program

Introduction

In this lab, you will build a Windows Forms application that provides a graphical user interface to the Simple Bank database.

Suggested Time: 45 minutes

Root Directory: OIC\IntroVb

Directories: Labs\Lab13A\BankGui (do your work here)
Chap13\BankGui\Step1 (backup of starter code)
Chap13\BankGui\Step2 (answer)

Database: OIC\Data\SimpleBank.mdf

Instructions

1. Build and run the starter program. Examine the code, and observe the following features.

   a. The class DB encapsulates all the ADO.NET code. In general, this is a good design, as opposed to having database code directly in the GUI. All the methods are shared, so it will not be necessary to instantiate a DB object.
b. The class **Account** encapsulates an account.

c. The file **Form1.vb** provides the user interface, with handlers for the Read, Previous and Next buttons. The Read button will read the database into a **List(Of Account)**, which can be traversed by the Previous and Next buttons. The number of accounts in the list will be shown in a status bar at the bottom of the form.

2. Provide a method **AddAccount()** in the **DB** class. It will be similar to the corresponding method in the **ConnectedSql()** program, but it will not do any output. Instead, it will return the number of rows updated to the calling program.

   ```vbnet
   Public Shared Function AddAccount(ByVal owner As String, ByVal bal As Decimal, ByVal atype As String) As Integer
   Dim query As String = "insert into Account " & _
   "(Owner, Balance, AccountType) values(" & owner & ", " & _
   bal & ", " & atype & ")"
   Dim command As IDbCommand = CreateCommand(query)
   conn.Open()
   Dim numrow As Integer = command.ExecuteNonQuery()
   conn.Close()
   Return numrow
   End Function
   ```

3. Provide a handler for the Add button in which you will call the **AddAccount()** method.

   ```vbnet
   Private Sub btnAdd_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnAdd.Click
   Dim owner As String = txtOwner.Text
   Dim balance As Decimal = Convert.ToDecimal(txtBalance.Text)
   Dim acctype As String = ""
   If radChecking.Checked Then
   acctype = "C"
   ElseIf radSavings.Checked Then
   acctype = "S"
   End If
   Dim numrow As Integer = DB.AddAccount(owner, balance, acctype)
   Dim msg As String = String.Format("{0} rows updated", numrow)
   MessageBox.Show(msg, "Simple Bank")
   End Sub
   ```

4. You may build and test at this point. Note that you do not have to specify an ID for a new account, as the database will generate an ID automatically for you. After you have added an account, you may click Read. You should see in the status bar that there is now one more account, and by clicking Next repeatedly, you will come to the new account.

5. In a similar manner add a method **RemoveAccount()** and use it to implement a handler for the Delete button. Finally, add a method **UpdateAccount()** and use it to implement a handler for the Update button. Build and test.
Lab 13B

Data Binding

Introduction

In this lab, you will enhance the data binding demo program to specify the CategoryId by means of a dropdown combobox, which is bound to the Category table of the DataSet. You can use the code generated by Visual Studio for the ToolStrip as a guide for your own code. When done, you can delete the ToolStrip.

[Suggested Time: 30 minutes]

Root Directory: OIC\IntroVb

Directories: Labs\Lab13B\ShowBooks (do your work here – starter code)
             Demos\ShowBooks (continue in-class demo)
             Chap13\ShowBooks\Step1 (backup of starter code)
             Chap13\ShowBooks\Step2 (answer)

Instructions

1. If you completed the in-class demo, you may continue working in the Demos\ShowBooks directory. Otherwise, you can use the starter code provided in the lab directory. Build and run your program to make sure it is working.

2. Drag a ComboBox onto the form. Change the name to cmbCat. Make sure that DropDownListStyle is DropDownList.

3. Open the smart tag. Check “Use Data Bound Items”. Several fields for Data Binding Mode open up.
4. For the Data Source, select the Category table in SmallPubDataSet, which you can find by opening up Other Data Sources and then Project Data Sources.

5. This will set up a binding through a new BindingSource, CategoryBindingSource. Display Member should be Description and Value Member should be CategoryId.

6. Build and run the program. You should see that the combobox is displaying the categories, but at this point it is not linked to populating the DataGridView control.
7. To see how we might invoke the query, examine the code generated by Visual Studio for clicking the BooksByCat button on the toolbar.

```vbnet
Me.bookTableAdapter.BooksByCat(Me.smallPubDataSet.Book, _
    (CInt(System.Convert.ChangeType(categoryIdToolStripTextBox.Text, _
        GetType(Integer)))))
```

8. The key is that the generated TableAdapter object has a method `BooksByCat()` for invoking the parameterized query. You simply have to pass as parameters the data set table and the CategoryId.

9. Double-click the combo box to add an event handler for the `SelectedIndexChanged` event. Provide the following code in a helper method `ShowBooks()`:

```vbnet
If cmbCat.SelectedIndex <> -1 Then
    Dim id As Integer = CInt(cmbCat.SelectedValue)
    bookTableAdapter.BooksByCat(smallPubDataSet.Book, id)
End If
```

10. Build and run. Both user interfaces for specifying a category should now be working. Delete the ToolStrip and clean up any stray code. Make sure that the program still works with your combo box.

11. Initially all the books are shown. To have only all the books of the current category shown initially, add a call to `ShowBooks()` in the Load event handler.

12. Build and exercise your program.