Object-Oriented Programming in C#

Evaluation Copy

Student Guide

Revision 4.5

Object Innovations Course 4101

Object-Oriented Programming in C#
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.

Authors: Robert J. Oberg, Howard Lee Harkness

Special Thanks: Marianne Oberg, Sharman Staples, Paul Nahay

Copyright ©2012 Object Innovations Enterprises, LLC. All rights reserved.

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

Printed in the United States of America on recycled paper.

Evaluation Copy

Table of Contents (Overview)

Evaluation Copy

Directory Structure

- **The course software installs to the root directory *C:\OIC\CSharp*.**
  - 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.
  - Supplementary course content is provided in PDF files in the **Supplements** directory. Code examples for the supplements are in directories **Supp1**, **Supp2**, **Supp3** and **Supp4**.

Evaluation Copy

Table of Contents (Detailed)

Evaluation Copy

Evaluation Copy

Evaluation Copy

Evaluation Copy

Evaluation Copy

Evaluation Copy

Evaluation Copy

Evaluation Copy

Evaluation Copy

Evaluation Copy

Evaluation Copy

Evaluation Copy

Chapter 1

.NET: What You Need to Know

Evaluation Copy

Copyright © 2012 Object Innovations Enterprises, LLC
All Rights Reserved

.NET: What You Need to Know

Objectives

After completing this unit you will be able to:

- **Describe the essentials of creating and running a program in the .NET environment.**
- **Build and run a simple C# program.**
- **Use the ILDASM tool to view intermediate language.**
- **Use Visual Studio 2012 as an effective environment for creating C# programs.**
- **Use the .NET Framework SDK documentation.**

Evaluation Copy

Copyright © 2012 Object Innovations Enterprises, LLC
All Rights Reserved

Getting Started

- **From a programmer's perspective, a beautiful thing about .NET is that you scarcely need to know anything about it to start writing programs for the .NET environment.**
  - You write a program in a high-level language (such as C#), 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.

Evaluation Copy

.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.

Evaluation Copy

.NET Programming in a Nutshell

1. Write your program in a high-level .NET language, such as C#.
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 C# program, and run it under .NET.**
  - Don’t worry about the syntax of C#, which we will start discussing in the next chapter.

Evaluation Copy

Copyright © 2012 Object Innovations Enterprises, LLC
All Rights Reserved

.NET Program Example

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

1. Enter the program in a text editor.

```
// SimpleCalc.cs
//
// This program does a simple calculation:
// calculate area of a rectangle

public class SimpleCalc
{
   static void Main()
   {
      int width = 20;
      int height = 5;
      int area;
      area = width * height;
      System.Console.WriteLine("area = {0}", area);
   }
}
```

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 folder \OIC\CSharp\Chap01\SimpleCalc.

```
>csc SimpleCalc.cs
```

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

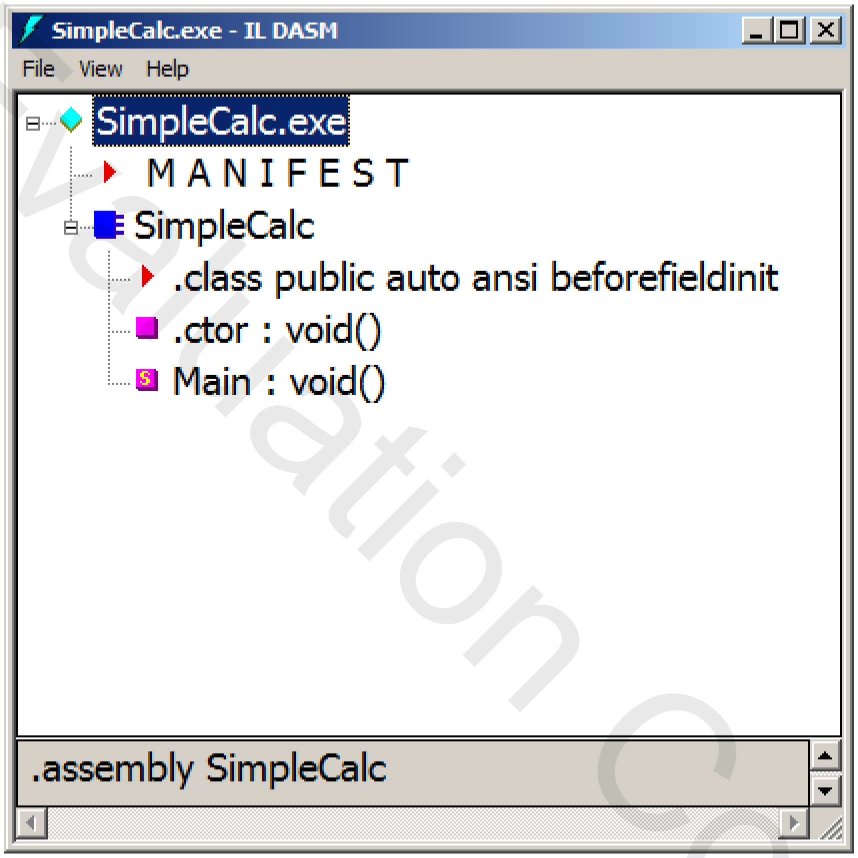
```
>SimpleCalc
area = 100
```

Evaluation Copy

Viewing the Assembly

- **You can view the assembly using the *ILDASM* tool[1].**

```
>ildasm SimpleCalc.exe
```



```
SimpleCalc.exe
  MANIFEST
  SimpleCalc
    .class public auto ansi beforefieldinit
    .ctor : void()
    Main : void()
```

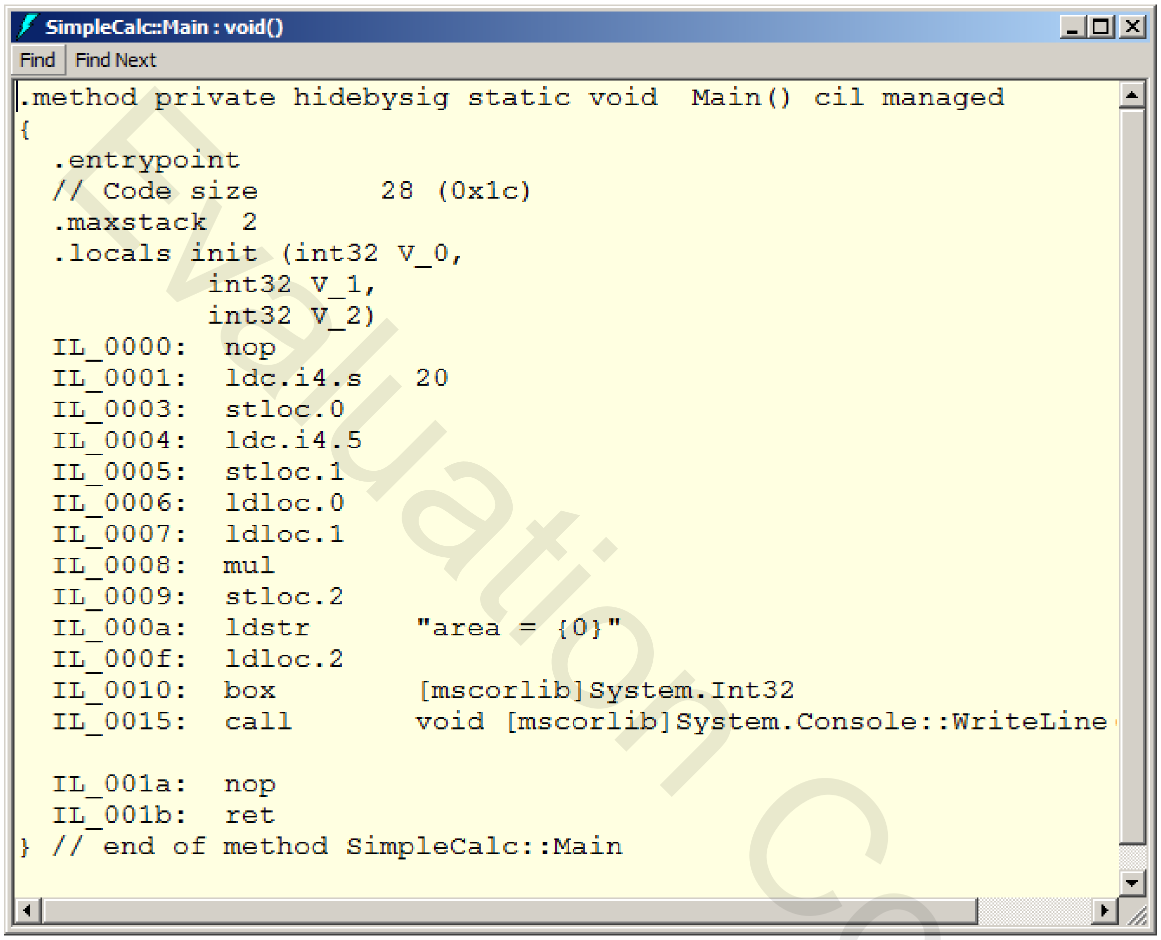
```
.assembly SimpleCalc
```

Evaluation Copy

[1] You can change the font size from the View menu.

Viewing Intermediate Language

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



```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       28 (0x1c)
  .maxstack  2
  .locals init (int32 V_0,
           int32 V_1,
           int32 V_2)
  IL_0000:  nop
  IL_0001:  ldc.i4.s   20
  IL_0003:  stloc.0
  IL_0004:  ldc.i4.5
  IL_0005:  stloc.1
  IL_0006:  ldloc.0
  IL_0007:  ldloc.1
  IL_0008:  mul
  IL_0009:  stloc.2
  IL_000a:  ldstr      "area = {0}"
  IL_000f:  ldloc.2
  IL_0010:  box        [mscorlib]System.Int32
  IL_0015:  call       void [mscorlib]System.Console::WriteLine

  IL_001a:  nop
  IL_001b:  ret
} // end of method SimpleCalc::Main
```

Evaluation Copy

Copyright © 2012 Object Innovations Enterprises, LLC
All Rights Reserved

Understanding .NET

- **The nice thing about a high-level programming language is that you usually do not need to be concerned with the platform on which the program executes.**
- **You can work with the abstractions provided by the language and with functions provided by libraries.**
- **Your appreciation of the C# 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:**
  - Test-Driven Development (Unit Testing)
  - .NET Framework Using C#
- **And then, depending on your interests:**

Data Access	**Windows**	**Web**
ADO.NET	Windows Forms	ASP.NET
XML Programming	WPF	ASP.NET MVC
		ASP.NET AJAX
		Silverlight
		WCF

Evaluation Copy

Visual Studio 2012

- **While it is possible to write C# 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 C# programs.**
- **Visual Studio 2012 is discussed in more detail in Supplement 1.**

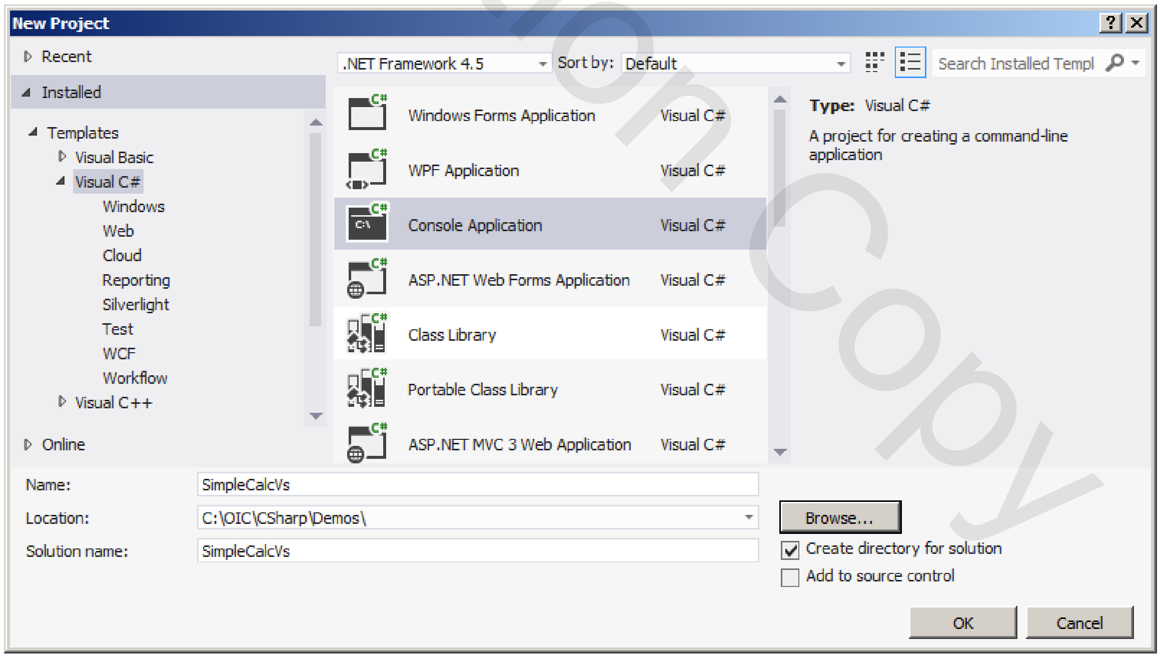
Evaluation Copy

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 Visual C# and "Console Application".
3. Leave .NET Framework 4.5 as the target framework. Leave checked "Create directory for solution". [2]
4. In the Name field, type **SimpleCalcVs** and for Location browse to **C:\OIC\CSharp\Demos**. Click OK.

Evaluation Copy



.NET Framework 4.5
Sort by: Default

Type: Visual C#
A project for creating a command-line application

Name: SimpleCalcVs
Location: C:\OIC\CSharp\Demos\
Solution name: SimpleCalcVs
[✓] Create directory for solution

[2] Examples in later chapters frequently do not have a directory for solution.

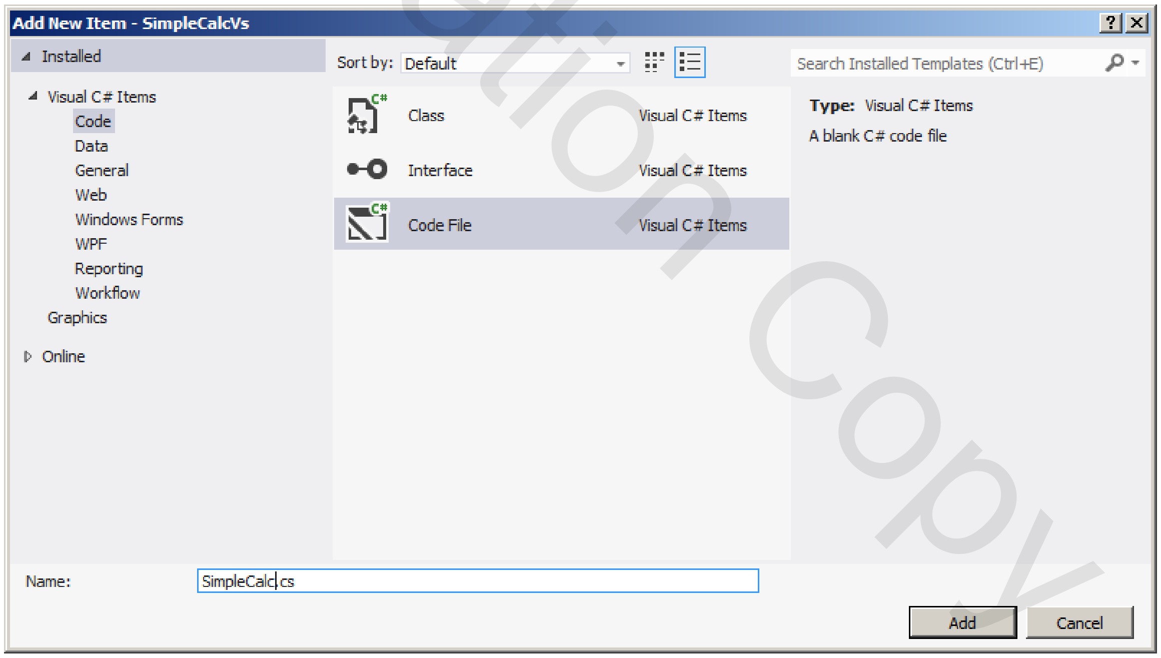
Copyright © 2012 Object Innovations Enterprises, LLC
All Rights Reserved

Adding a C# File

- **There will be a number of starter files. Expand properties and select the files *AssemblyInfo.cs*, *App.config* and *Program.cs*. Press the Delete key.**
- **We are now going to add a file *SimpleCalc.cs*, which contains the text of our program.**

1. In Solution Explorer, right click over **SimpleCalcVs** and choose Add | New Item.... This will bring up the Add New Item dialog.
2. In the middle pane, choose "Code File." For Name type **SimpleCalc.cs**. Click Add.

Evaluation Copy



Type: Visual C# Items
A blank C# code file
Name: SimpleCalc.cs

Using the Visual Studio Text Editor

- **The empty file *SimpleCalc.cs* will now be open in the Visual Studio text editor. Enter the following program.**
  - Or you could just copy from **Chap01\SimpleCalc\**.

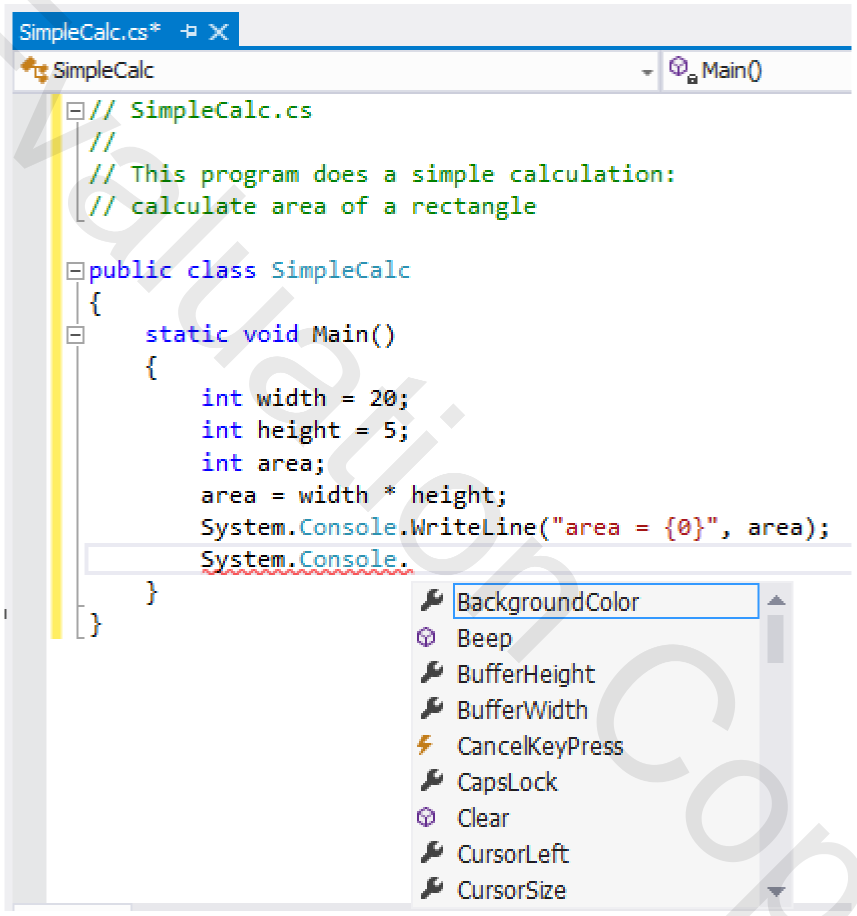
```
// SimpleCalc.cs
//
// This program does a simple calculation:
// calculate area of a rectangle

public class SimpleCalc
{
   static void Main()
   {
      int width = 20;
      int height = 5;
      int area;
      area = width * height;
      System.Console.WriteLine("area = {0}", area);
   }
}
```

  - Notice that the Visual Studio text editor highlights syntax, indents automatically, and so on.

Evaluation Copy

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.



```
// SimpleCalc.cs
//
// This program does a simple calculation:
// calculate area of a rectangle

public class SimpleCalc
{
    static void Main()
    {
        int width = 20;
        int height = 5;
        int area;
        area = width * height;
        System.Console.WriteLine("area = {0}", area);
        System.Console.
    }
}
```

Evaluation Copy

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:**
  - Menu Debug | Start Without Debugging
  - Toolbar button (This 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 button Start
  - Keyboard shortcut F5.

Evaluation Copy

[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.**

```
public class SimpleCalc
{
   static void Main()
   {
      int width = 20;
      int height = 5;
      int area;
      area = width * height;
      System.Console.WriteLine("area = {0}", area);
      System.Console.WriteLine(
         "Prese Enter to exit");
      System.Console.ReadLine();
   }
}
```

- **This program is saved as a Visual Studio solution in *Chap01\SimpleCalcVs*[4].**
- **Remember that you can always make the console window stay open by running without the debugger via Ctrl + F5.**

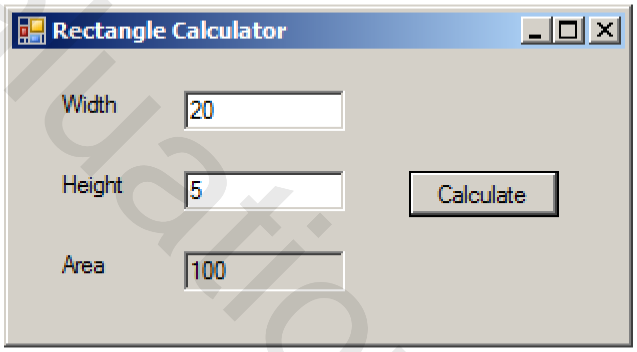
Evaluation Copy

[4] The solution can be opened in either Visual Studio 2010 or Visual Studio 2012. The project uses .NET 4.0 and so will run on either .NET 4.0 or .NET 4.5. The same is true with Chap01\SimpleCalcGui.

Visual C# and GUI Programs

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

  <u>Example</u>: See **Chap01\SimpleCalcGui**



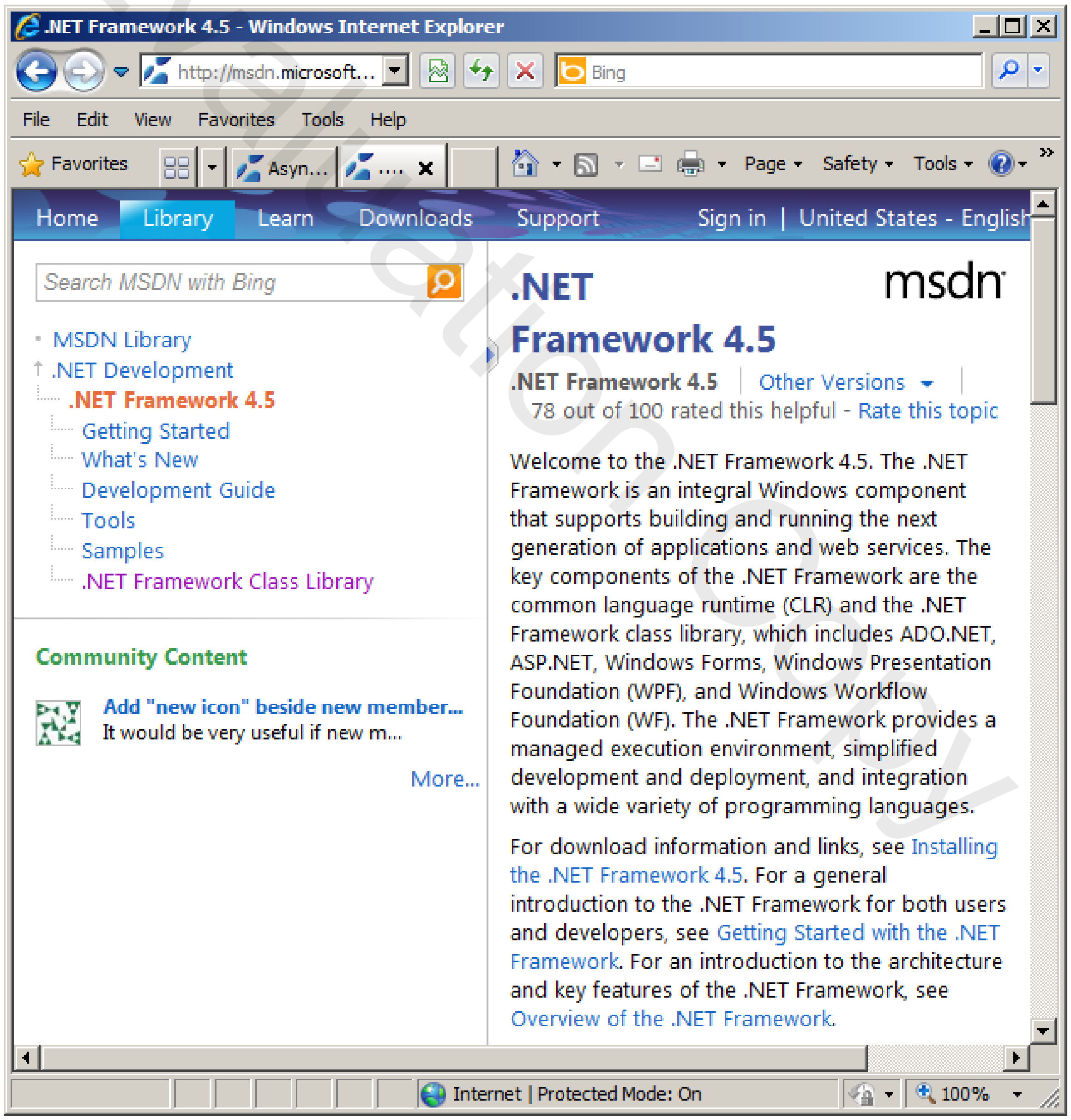
- **We will discuss GUI programming using C# in Chapter 6.**

Evaluation Copy

.NET Documentation

- **.NET Framework documentation 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.

Evaluation Copy


.NET Framework 4.5 - Windows Internet Explorer
http://msdn.microsoft...
Bing
File Edit View Favorites Tools Help
Favorites
Asyn...
Page Safety Tools
Home Library Learn Downloads Support
Sign in | United States - English
Search MSDN with Bing
msdn
Internet | Protected Mode: On
100%

MSDN Library
.NET Development
.NET Framework 4.5
Getting Started
What's New
Development Guide
Tools
Samples
.NET Framework Class Library

Community Content

Add "new icon" beside new member...
It would be very useful if new m...

More...

.NET Framework 4.5

.NET Framework 4.5 | Other Versions

78 out of 100 rated this helpful - Rate this topic

Welcome to the .NET Framework 4.5. The .NET Framework is an integral Windows component that supports building and running the next generation of applications and web services. The key components of the .NET Framework are the common language runtime (CLR) and the .NET Framework class library, which includes ADO.NET, ASP.NET, Windows Forms, Windows Presentation Foundation (WPF), and Windows Workflow Foundation (WF). The .NET Framework provides a managed execution environment, simplified development and deployment, and integration with a wide variety of programming languages.

For download information and links, see Installing the .NET Framework 4.5. For a general introduction to the .NET Framework for both users and developers, see Getting Started with the .NET Framework. For an introduction to the architecture and key features of the .NET Framework, see Overview of the .NET Framework.

Copyright © 2012 Object Innovations Enterprises, LLC
All Rights Reserved

Summary

- **As in other environments, with .NET you write a program in a high-level language, compile to an executable (.EXE file), and run that .EXE file.**
- **The .EXE file, called an *assembly*, contains Intermediate Language instructions.**
- **You can view an assembly through the *ILDASM* tool.**
- **Visual Studio 2012 is a powerful IDE that makes it easy to develop C# programs.**
- **With Visual Studio it is easy to create GUI programs using C#.**
- **You can access extensive .NET Framework documentation through the Visual Studio help system.**

Evaluation Copy

Copyright © 2012 Object Innovations Enterprises, LLC
All Rights Reserved

Evaluation Copy

Copyright © 2012 Object Innovations Enterprises, LLC
All Rights Reserved

Chapter 5

Control Structures

Evaluation Copy

Control Structures

Objectives

After completing this unit you will be able to:

- **Use the common C# control structures to perform tests and loops.**
- **Use arrays in C# programs.**

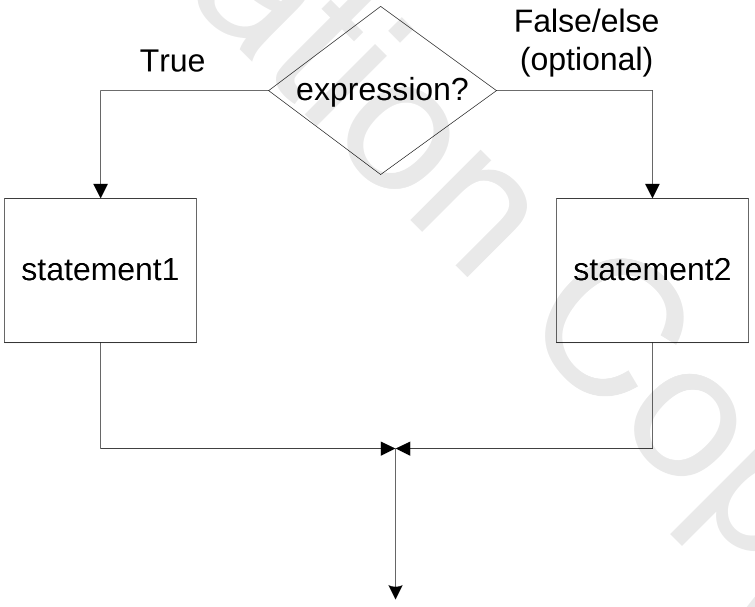
Evaluation Copy

If Test

- **In an *if test*, a *bool* expression is evaluated, and, depending on the result, the "true branch" or "false branch" is executed.**

```
if (expression)
    statement 1;
else // optional
    statement 2;
```

- **If the *else* is omitted, and the test is false, the control simply passes to the next statement after the *if* test.**



  – See **LeapYear**.

Evaluation Copy

Blocks

- **Several statements may be combined into a *block*, which is semantically equivalent to a single statement.**
  - A block is enclosed in curly braces.
  - Variables declared inside a block are local to that block.
- **The program *Swap* illustrates a block and the declaration of a local variable *temp* within the block.**
  - An attempt to use **temp** outside the block is a compiler error.

```
// Swap.cs

using System;

public class Swap
{
   public static int Main(string[] args)
   {
      int x = 5;
      int y = 12;
      Console.WriteLine("Before: x = {0}, y = {1}",
                        x, y);
      if (x < y)
      {
         int temp = x;
         x = y;
         y = temp;
      }
      Console.WriteLine("After: x = {0}, y = {1}",
                        x, y);
      // Console.WriteLine("temp = {0}", temp);
      return 0;
   }
}
```

Evaluation Copy

Loops

- **while**
- **for**
- **do/while**
- **foreach**
- **break**
- **continue**
- **goto**
- **switch**

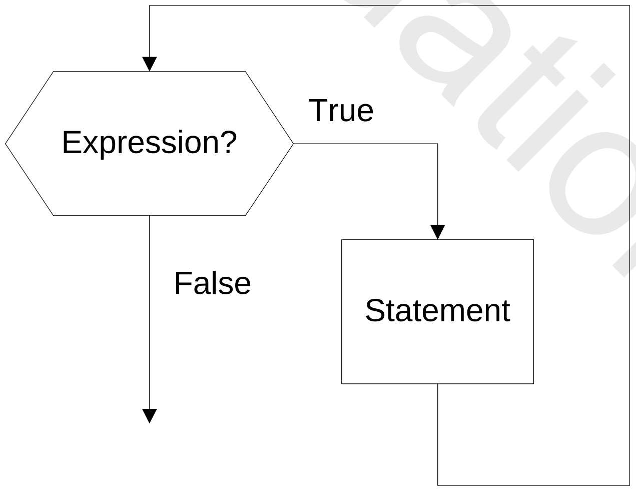
Evaluation Copy

while Loop

- **The most basic type of loop in C# is a *while* loop.**

```
while (expression)
{
    statements;
    ...
}
more statements;
```

  - Recommendation: Use blocks (in curly braces) even if there is only one statement in a loop.



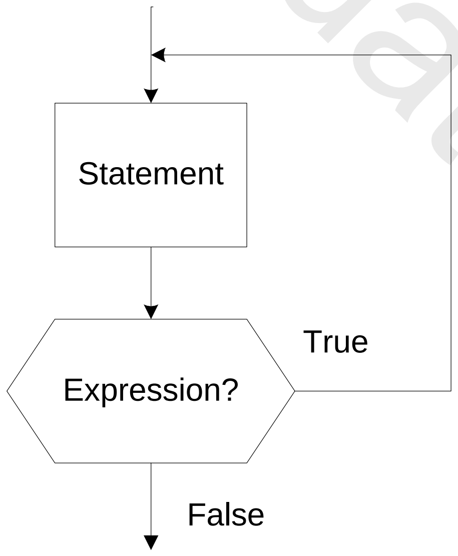
  - See **LeapYearLoop**.

Evaluation Copy

do/while Loops

- **In the while loop, if the condition is initially false, then the loop is skipped.**
- **If you want a loop in which the body is always executed, use a do/while.**

```
do
{
  ...
}
while (expression); // ← note semicolon!
```



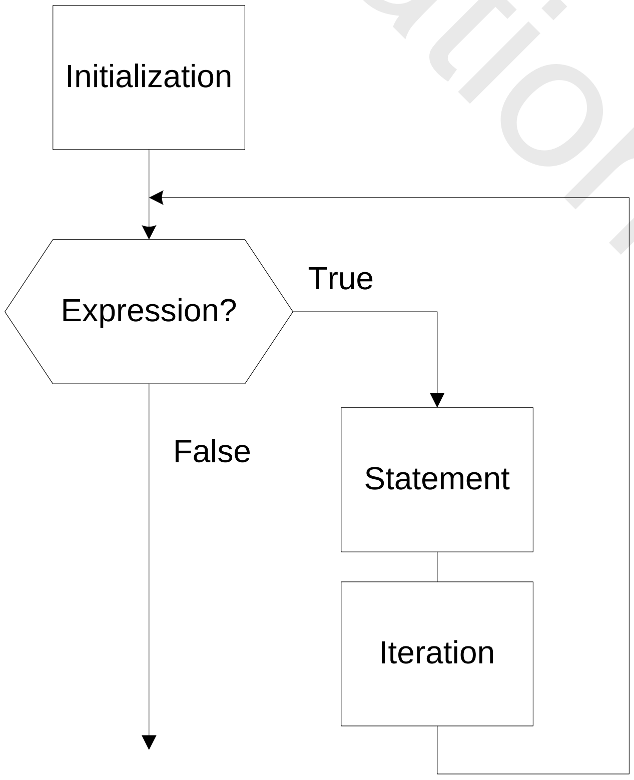
Evaluation Copy

for Loops

- **A perennial favorite of C/C++ and Java programmers, the *for* loop is the most flexible of the loop control structures.**

```
for (initialization; test; iteration)
{
  statements;
  ...
}
more statements;
```

  – The test must be a Boolean expression. Initialization and iteration can be almost any kind of expression.



Evaluation Copy

ForUp Example

- **The example program *ForUp* illustrates calculating the sum of the numbers from 1 to 100 using a for loop with the counter going up.**
  - Notice that in this loop the variable **i** is defined within the loop and hence is not available outside the loop.

```
// ForUp.cs

using System;

public class ForUp
{
   public static int Main(string[] args)
   {
      int sum = 0;
      for (int i = 1; i <= 100; i++)
      {
         sum += i;
      }
      Console.WriteLine("sum = {0}", sum);
      // Console.WriteLine("i = {0}", i);
      // i is not defined outside the for loop
      return 0;
   }
}
```

Evaluation Copy

ForDown Example

- **The example *ForDown* illustrates calculating the sum of the numbers from 1 to 100 using a for loop with the counter going down.**
  - Notice that in this loop the variable **i** is defined before the loop and hence is available outside the loop.

```
// ForDown.cs

using System;

public class ForDown
{
   public static int Main(string[] args)
   {
      int sum = 0;
      int i;
      for (i = 100; i >= 1; i--)
      {
         sum += i;
      }
      Console.WriteLine("sum = {0}", sum);
      Console.WriteLine("i = {0}", i);
      // i is defined outside the for loop
      return 0;
   }
}
```

Evaluation Copy

Arrays

- **Arrays are a very common and easy to use data structure in many programming languages, and they are useful for illustrating programs involving loops.**
  - Hence we will give a brief preview here, so that we can provide more interesting examples for the rest of the chapter.
- **An array is declared using square brackets [] after the type, not after the variable.**

```
int [] a;    // declares an array of int
```

  - Note that the size of the array is not part of its type. The variable declared is a reference to the array.
  - You create the array elements and establish the size of the array using the **new** operator.

```
a = new int[10];     // creates 10 array elements
```

  - The new array elements start out with the appropriate default values for the type (0 for **int**).
  - You may both declare and initialize array elements using curly brackets, as in C/C++.

```
int [] a = {2, 3, 5, 7, 11};
```

Evaluation Copy

Fibonacci Example

- **As our first example we will populate a 10-element array with the first 10 Fibonacci numbers. The Fibonacci sequence is defined as follows:**

```
fib[0] = 1
fib[1] = 1
fib[i] = fib[i-1] + fib[i-2] for i >= 2
```

  – The program **Fibonacci** populates the array and then prints out the first 10 Fibonacci elements all on one line, followed by printing them out in reverse order on the next line.

```
int [] fib;
fib = new int[10];
fib[0] = fib[1] = 1;
for (int i = 2; i < 10; i++)
   fib[i] = fib[i-1] + fib[i-2];

for (int i = 0; i < 10; i++)
   Console.Write("{0} ", fib[i]);
Console.WriteLine();

for (int i = 9; i >= 0 ; i--)
   Console.Write("{0} ", fib[i]);
Console.WriteLine();
```

- **Here is the output:**

```
1 1 2 3 5 8 13 21 34 55
55 34 21 13 8 5 3 2 1 1
```

Evaluation Copy

foreach Loop

- **The *foreach* loop is familiar to VB programmers, but is not present in C/C++ or Java (before Java 5).**
- **It is a special loop for iterating through collections.**
- **In C#, an array is a collection, so you can use a *foreach* loop to iterate through an array.**

```
// ForEachLoop.cs

using System;

public class ForEachLoop
{
   public static int Main(string[] args)
   {
      int [] primes = {2, 3, 5, 7, 11, 13};
      int sum = 0;
      foreach (int prime in primes)
      {
         Console.Write("{0} ", prime);
         sum += prime;
      }
      Console.WriteLine();
      Console.WriteLine("sum = {0}", sum);
      return 0;
   }
}
```

- ***foreach* will be covered in greater detail in a later chapter.**

Evaluation Copy

break

- **The *break* statement allow immediate exit from a loop.**
  - See **BreakSearch.**

```
int [] primes = {2, 3, 5, 7, 11, 13};
foreach (int prime in primes)
   Console.Write("{0} ", prime);
Console.WriteLine();
int target = 7;
int i;
for(i = 0; i < primes.Length; i++)
{
   if (target == primes[i])
      break;
}
if (i == primes.Length)
   Console.WriteLine("{0} not found", target);
else
   Console.WriteLine("{0} found at {1}", target,i);
return 0;
```

- **Here is the output:**

```
2 3 5 7 11 13
7 found at 3
```

Evaluation Copy

continue

- **The *continue* statement bypasses the remainder of a loop, transferring control to the beginning of the loop.**
  - See **ContinueLoop.**

```
int [] numbers = {0,1,2,3,4,5,6,7,8,9};
foreach(int num in numbers)
{
   Console.Write("{0} ", num);
}
Console.WriteLine();
Console.Write("Odd numbers: ");
int index = 0;
while(++index < numbers.Length)
{
   if(numbers[index] % 2 == 0)
   {
      continue;
   }
   Console.Write("{0} ", numbers[index]);
}
Console.WriteLine();
```

- **Here is the output:**

```
0 1 2 3 4 5 6 7 8 9
Odd numbers: 1 3 5 7 9
```

Evaluation Copy

goto

- **Considered by purists to be evil, the infamous *goto* was even completely banned from some languages.**
  - Use **goto** sparingly and with great care.

```
goto label;
  ...
label:
  ...
```

```
// GotoSearch.cs
using System;
public class GotoSearch
{
   public static int Main(string[] args)
   {
      int [] primes = {2, 3, 5, 7, 11, 13};
      foreach (int prime in primes)
         Console.Write("{0} ", prime);
      Console.WriteLine();
      int target = 7;
      int i;
      for(i = 0; i < primes.Length; i++)
      {
         if (target == primes[i])
            goto found;
      }
      Console.WriteLine("{0} not found", target);
      return 0;
   found:
      Console.WriteLine("{0} found at {1}",
                        target,i);
      return 0;
   }
}
```

Evaluation Copy

Structured Programming

- **Although a program like the one shown in the preceding page is easy to understand on a small scale, the structure of such a program is problematical if the same style is carried over to larger programs.**
  - The basic difficulty is that there are many execution paths, and so it becomes difficult to verify that the program is correct.
- ***Structured programming* imposes certain discipline.**
  - Programs are built out of basic components, such as blocks (compound statements) and simple control structures like **if...else** and **while**.
  - Each of these components has a single entrance and a single exit.
- **The program on the preceding page violates these principle several places.**
  - The **Main** function has two exits (return statements).
  - The loop can be exited in two ways, normally and via the goto.
  - Such a program can become difficult to maintain. If some task needs to always be done before exiting a loop, you may have to place duplicate code, which can become out of synch when this common code is updated in one place.

Evaluation Copy

Copyright © 2012 Object Innovations Enterprises, LLC
All Rights Reserved

Structured Search Example

- **The program *StructuredSearch* illustrates a more structured programming approach to our simple linear search than either of our previous solutions.**
  - Both **break** and **goto** can be replaced by a simple while loop and use of a suitable **bool** flag.

```
int [] primes = {2, 3, 5, 7, 11, 13};
foreach (int prime in primes)
   Console.Write("{0} ", prime);
Console.WriteLine();
int target = 7;
int i = 0;
bool found = false;
while (!found && i < primes.Length)
{
   if (target == primes[i])
      found = true;
   else
      i++;
}
if (found)
   Console.WriteLine("{0} found at {1}", target,i);
else
   Console.WriteLine("{0} not found", target);
return 0;
```

- **Here is the output:**

```
2 3 5 7 11 13
7 found at 3
```

Evaluation Copy

Multiple Methods

- **Our example programs so far have all of our code in one *Main()* method.**
- **As programs get longer, use subroutines, or additional "methods" in C# terminology.**
  - For now, look at the example **MultipleMethods**.

```
using System;

public class MultipleMethods
{
   public static void Main()
   {
      InputWrapper iw = new InputWrapper();
      // initialize and display array
      int[] primes = {2, 3, 5, 7, 11, 13};
      for (int i = 0; i < primes.Length; i++)
         Console.Write("{0} ", primes[i]);
      Console.WriteLine();
      // loop to read and search for targets
      Console.WriteLine(
"Enter numbers to search for, -1 when done");
      int target = iw.getInt("target number: ");
      while (target != -1)
      {
         int index = Search(primes, target);
         if (index == -1)
            Console.WriteLine(
               "{0} not found", target);
         else
            Console.WriteLine(
               "{0} found at {1}", target,index);
         target = iw.getInt("target number: ");
      }
   }
```

Evaluation Copy

Multiple Methods (Cont'd)

```
   public static int Search(int[] array,
                            int target)
   {
      int i = 0;
      bool found = false;
      while (!found && i < array.Length)
      {
         if (target == array[i])
            found = true;
         else
            i++;
      }
      if (found)
         return i;
      else
         return -1;
   }
}
```

- **Here is a sample run of this program:**

```
2 3 5 7 11 13
Enter numbers to search for, -1 when done
target number: 11
11 found at 4
target number: 3
3 found at 1
target number: 33
33 not found
target number: 13
13 found at 5
target number: 2
2 found at 0
target number: -1
Press any key to continue
```

Evaluation Copy

switch

- **The *switch* statement can be substituted, in some cases, for a sequence of *if* tests.**
- **There are comparable control structures in other languages, such as:**
  - **Select** in Visual Basic
  - **case** in Pascal
  - "computed goto" in FORTRAN.
  - **switch** in C/C++
- **Example Program:**
  - **SwitchDemo**

Evaluation Copy

switch in C# and C/C++

- **In C#, after a particular case statement is executed, control does not automatically continue to the next statement.**
  - You must explicitly specify the next statement, typically by a **break** or **goto** *label*.
  - This avoids a "gotcha" in C/C++.

```
switch (code)
{
  case 1:
    goto case 2;
  case 2:
    Console.WriteLine("Low");
    break;
  case 3:
    Console.WriteLine("Medium");
    break;
  case 4:
    Console.WriteLine("High");
    break;
  default:
    Console.WriteLine("Special case");
    break;
}
```

- **In C#, you may switch on any integer type and on a *char* or *string* data type.**

Evaluation Copy

Lab 5

Managing a List of Contacts

In this lab, you will begin implementation of a contact management system. The first version of the program is very simple. You will maintain a list of names in an array of strings, and you will provide a set of commands to work with these contacts:

- add a contact to the list
- show the contacts in forward order
- show the contacts in reverse order
- find a contact in the list.
- remove a contact from the list

Detailed instructions are contained in the Lab 5 write-up at the end of the chapter.

Suggested time: 45 minutes

Evaluation Copy

Summary

- **C# has a variety of control structures of C#, including *if*, *while*, *do*, *for* and *switch*.**
- **There are alternative ways of exiting or continuing iteration in a loop, including *break*, *continue*, and *goto*.**
- **Structured programming avoids use of *goto* and leads to programs that are easier to understand and maintain.**
- **C# provides arrays for holding collections of items all of the same type.**
- **The *foreach* loop makes it very easy to write concise code for iterating through an array or another collection.**

Evaluation Copy

Lab 5

Managing a List of Contacts

Introduction

In this lab, you will begin implementation of a contact management system. The first version of the program is very simple. You will maintain a list of names in an array of strings, and you will provide a set of commands to work with these contacts:

- add a contact to the list
- show the contacts in forward order
- show the contacts in reverse order
- find a contact in the list.
- remove a contact from the list

Evaluation Copy

Suggested Time: 45 minutes

Root Directory: **OIC\CSharp**

Directories:	**Labs\Lab5\Contacts**	(do your work here)
	Chap05\Contacts\Step1	(answer to Part 1)
	Chap05\Contacts\Step2	(answer to Part 2)

Part 1. Implement a Command Processing Loop

1. Use Visual Studio to create an empty C# project **Contacts** in the **Lab5** folder. This will create the subfolder **Contacts**. Add a new file **TestContacts.cs** to your project, where you will place the program code.

2. Move the supplied file **InputWrapper.cs** from **Lab5** down to **Lab5\Contacts**. Add this file to your project.

3. Add C# code to the file **TestContacts.cs** to set up a class **TestContacts** with a public static **Main()** method. Provide a **using System;** statement.

4. Add code to **Main()** to do the following:

    a. Instantiate an **InputWrapper** object **iw**.

    b. Write a message “Enter command, quit to exit.”

    c. Use the **getString()** method of **InputWrapper** to prompt for a command using the prompt string “> “ and store the result in the **string** variable **cmd**.

    d. Write out the command that was entered.

5. Build and test. It would be a good idea to also build and test incrementally after the following steps, but we won't explicitly say so.

6. Add a while loop that will loop until the command entered is "quit." Move the statement writing out the command inside the loop.

7. Add a switch statement, with cases for each of the supported commands. In the default case, print out a message listing each of the legal commands with a brief description. In the case for a command provide stub code that prints out a message indicating that that command was invoked. You can now comment out the statement writing out the command that was entered.

8. Build and test.

Part 2. Implement the Commands

In this part you will declare an array of strings to hold the names. You will provide code for each of the commands, commenting out the stub code as each command is implemented.

1. Declare an array **names** of 10 strings. Also declare an **int** variable **count**, which will be initialized at 0. This holds a count of the actual number of elements in the array.

2. Add code to initialize a few names in the array. Increment **count** as you add each name. For example, the following code would add three names:

```
names[count++] = "Tom";
names[count++] = "Dick";
names[count++] = "Harry";
```

3. Implement the "forward" command to display the names in the array. First try a **foreach** loop. What is the problem?

4. The **foreach** loop will try to display all 10 elements in the array, and you want to display only the three actual names. Replace the **foreach** loop by a counted **for** loop, with the loop index incrementing.

5. Implement the "backward" command. Use a counted **for** loop, with the index decrementing.

6. Implement the "add" command.

7. Implement the "find" command. You will need search code both for this command and also for "remove," so it would be useful for you to create a method **Search()**, similar to the method in the **MultipleSearch** example program.

8. Implement the "remove" command.

Evaluation Copy

Chapter 9

Methods, Properties, and Operators

Evaluation Copy

Methods, Properties, and Operators

Objectives

After completing this unit you will be able to:

- **Explain how methods are defined and used, how parameters are passed to and from methods, and how the same method name can be overloaded, with different versions having different parameter lists.**
- **Implement methods in C# that take a variable number of parameters.**
- **Use the C# get/set (property syntax) methods for accessing data.**
- **Overload operators in C#, making the invocation of certain methods more natural and intuitive.**

Evaluation Copy

Static and Instance Methods

- **We have seen that classes can have different kinds of members, including fields, constants, and *methods*.**
  - A method implements behavior that can be performed by an object or a class.
  - Ordinary methods, sometimes called **instance methods**, are invoked through an object instance.

```
Account acc = new Account();
acc.Deposit(25);
```

  - Static methods are invoked through a class and do not depend upon the existence of any instances.

```
int sum = SimpleMath.Add(5, 7);
```

Evaluation Copy

Method Parameters

- **Methods have a list of parameters, which may be empty.**
  - Methods either return a value or have a **void** return.
  - Multiple methods may have the same name, so long as they have different signatures (a feature known as **method overloading)**.
  - Methods have the same signature if they have the same number of parameters and these parameters have the same types and modifiers (such as **ref** or **out**).
- **The return type does not contribute to defining the signature of a method. By default, parameters are value parameters, meaning copies are made of the parameters.**
  - The keyword **ref** designates a **reference** parameter, in which case, the parameter inside the method and the corresponding actual argument refer to the same object.
  - The keyword **out** refers to an **output** parameter, which is the same as a reference parameter, except that on the calling side, the parameter need not be assigned prior to the call.
  - We will study parameter passing and method overloading in more detail later in this chapter.

Evaluation Copy

No “Freestanding” Functions in C#

- **In C#, *all* functions are methods and, therefore, associated with a class.**
  - There is no such thing as a freestanding function, as in C and C++.
  - “All functions are methods” is rather similar to “everything is an object” and reflects the fact that C# is a pure object-oriented language.
  - The advantage of all functions being methods is that classes become a natural organizing principle. Methods are nicely grouped together.

Evaluation Copy

Classes with All Static Methods

- **Sometimes part of the functionality of your system may not be tied to any data, but may be purely functional in nature.**
- **In C#, you would organize such functions into classes that have all static methods and no fields.**
- **The program *TestSimpleMath/Step1* provides an elementary example.**

```
// SimpleMath.cs

public class SimpleMath
{
   public static int Add(int x, int y)
   {
      return x + y;
   }
   public static int Multiply(int x, int y)
   {
      return x * y;
   }
}
```

Evaluation Copy

Parameter Passing

- **Programming languages have different mechanisms for passing parameters.**
- **In the C family of languages, the standard is “call-by-value.”**
  - This means that the actual data values themselves are passed to the method.
  - Typically, these values are pushed onto the stack and the called function obtains its own independent copy of the values.
  - Any changes made to these values will not be propagated back to the calling program. C# provides this mechanism of parameter passing as the default, but C# also supports reference parameters and output parameters.
  - In this section, we will examine all three of these mechanisms, and we will look at the ramifications of passing class and struct data types.

Evaluation Copy

Parameter Terminology

- **Storage is allocated on the stack for method parameters.**
  - This storage area is known as the **activation record**.
  - It is popped when the method is no longer active.
  - The **formal parameters** of a method are the parameters as seen within the method.
  - They are provided storage in the activation record.
  - The **arguments** of a method are the expressions between commas in the parameter list of the method call.

```
int sum = SimpleMath.Add(5, 7);
                          // actual parameters are
                          // 5 and 7
...

public static int Add(int x, int y)
{                         // formal parameters are
                          // x and y
   ...
}
```

Evaluation Copy

Value Parameters

- **Parameter passing is the process of initializing the storage of the formal parameters by the actual parameters.**
- **The default method of parameter passing in C# is *call-by-value*, in which the values of the actual parameters are copied into the storage of the formal parameters.**
  - Call-by-value is safe, because the method never directly accesses the actual parameters, only its own local copies.
- **But there are drawbacks to call-by-value:**
  - There is no direct way to modify the value of an argument. You may use the return type of the method, but that only allows you to pass one value back to the calling program.
  - There is overhead in copying a large object.
- **The overhead in copying a large object is borne when you pass a *struct* instance.**
  - If you pass a class instance, or an instance of any other reference type, you are passing only a reference and not the actual data itself.
  - This may sound like call-by-reference, but what you are actually doing is passing a reference by value.
  - Later in this section, we will discuss the ramifications of passing struct and class instances.

Evaluation Copy

Copyright © 2012 Object Innovations Enterprises, LLC
All Rights Reserved

Reference Parameters

- **Consider a situation in which you want to pass more than one value back to the calling program.**
- **C# provides a clean solution through *reference parameters*.**
  - You declare a reference parameter with the **ref** keyword, which is placed before both the formal parameter and the actual parameter.
  - A reference parameter does not result in any copying of a value.
  - Instead, the formal parameter and the actual parameter refer to the same storage location.
  - Thus, changing the formal parameter will result in the actual parameter changing, as both are referring to exactly the same storage location.

Evaluation Copy

Copyright © 2012 Object Innovations Enterprises, LLC
All Rights Reserved

Reference Parameters (Cont’d)

- **The program *ReferenceMath* illustrates using *ref* parameters.**
  - The two methods **Add** and **Multiply** are replaced by a single method **Calculate**, which passes back two values as reference parameters.

```
// ReferenceMath.cs

public class ReferenceMath
{
   public static void Calculate(int x, int y,
                       ref int sum, ref int prod)
   {
      sum = x + y;
      prod = x * y;
   }
}
```

Evaluation Copy

Reference Parameters (Cont’d)

- **Notice the use of the *ref* keyword in front of the third and fourth parameters. Here is the test program:**

```
// TestReferenceMath.cs

using System;

public class TestReferenceMath
{
   public static void Main(string[] args)
   {
      int sum = 0, product = 0;
      MultipleMath.Calculate(5, 7, ref sum,
         ref product);
      Console.WriteLine("sum = {0}", sum);
      Console.WriteLine("product = {0}", product);
   }
}
```

- **The *ref* keyword is used in front of the parameters.**
- **Variables must be initialized before they are used as reference parameters.**

Evaluation Copy

Copyright © 2012 Object Innovations Enterprises, LLC
All Rights Reserved

Output Parameters

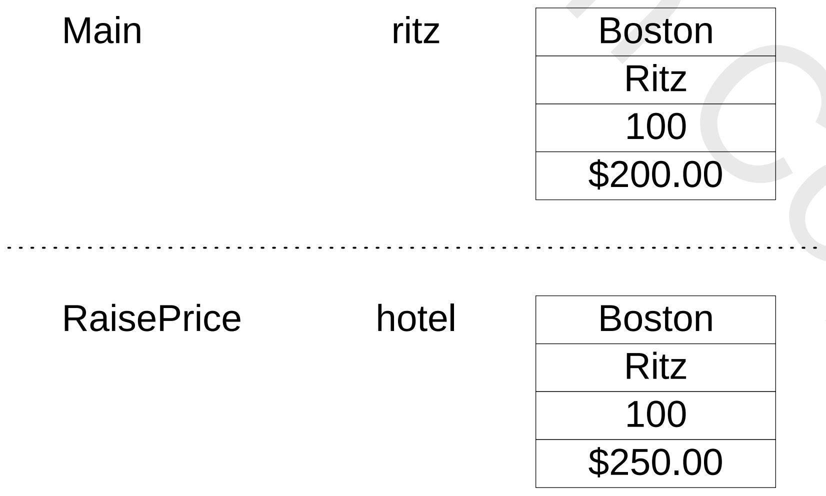
- **A reference parameter is used for two-way communication between the calling program and the called program, both passing data in and getting data out.**
- **Thus, reference parameters must be initialized before use.**
  - In **TestReferenceMath.cs** , we are only obtaining output, so initializing the variables only to assign new values is rather pointless.
  - C# provides for this case with **output parameters**.
  - Use the keyword **out** wherever you would use the keyword **ref**.
  - Then you do not have to initialize the variable before use.
  - Naturally, you could not use an **out** parameter inside the method; you can only assign it.
- **The program *OutputMath* illustrates the use of output parameters.**

```
public static void Calculate(int x, int y,
   out int sum, out int prod)     // definition
...

int sum, product;             // no initialization
OutputMath.Calculate(5, 7, out sum, out product);
                              // use
```

Evaluation Copy

Structure Parameters

- **A struct is a value type, so that if you pass a struct as a value parameter, the struct instance in the called method will be an independent copy of the struct in the calling method.**
- **The program *HotelStruct* illustrates passing an instance of a *Hotel* struct by value.**
- **The object *hotel* in the *RaisePrice* method is an independent copy of the object *ritz* in the Main method.**
  - This figure shows the values in both structures after the price has been raised for **hotel**.
  - Thus, the change in price does not propagate back to **Main**.



  - The program **HotelStructRef** has the same struct definition, but the test program passes the **Hotel** instance by reference.
  - Now the change does propagate, as you would expect.

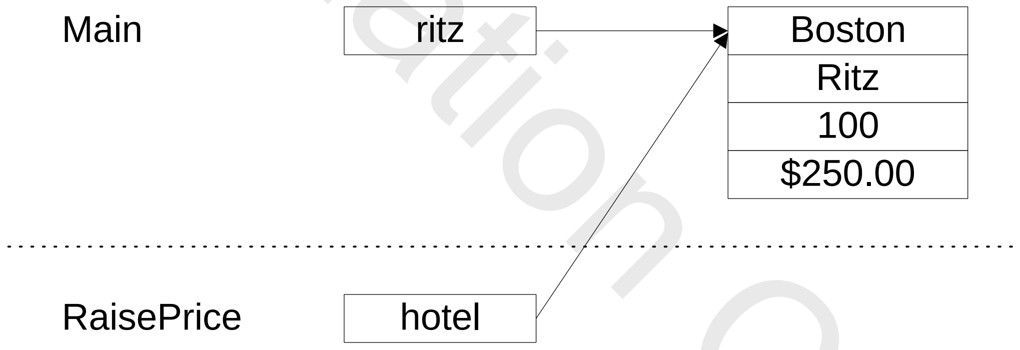
Evaluation Copy

Copyright © 2012 Object Innovations Enterprises, LLC
All Rights Reserved

Class Parameters

- **A class is a reference type, so that if you pass a class instance as a value parameter, the class instance in the called method will refer to the same object as the reference in the calling method.**

- **The program *HotelClass/Step1* illustrates passing an instance of a *Hotel* class by value.**
  - This figure illustrates how the **hotel** reference in the **RaisePrice** method refers to the same object as the **ritz** reference in **Main**.



  - Thus, when you change the price in the **RaisePrice** method, the object in **Main** is the same object and shows the new price.

Evaluation Copy

Method Overloading

- **In a traditional programming language, such as C, you need to create unique names for all of your methods.**
- **If methods basically do the same thing, but only apply to different data types, it becomes tedious to create unique names.**
  - For example, suppose you have a **FindMax** method that can find the maximum of two **int,** two **long,** or two **string**.
  - If we need to come up with a unique name for each method, we would have to create method names, such as **FindMaxInt**, **FindMaxLong**, and **FindMaxString**.
- **In C#, as in other object-oriented languages such as C++ and Java, you may *overload* method names.**
  - That is, different methods can have the same name, if they have different **signatures**.
  - Two methods have the same signature if they have the same number of parameters, the parameters have the same data types, and the parameters have the same modifiers (none, **ref**, or **out**).
  - The return type does not contribute to defining the signature of a method.
  - So, in order to have two functions with the same name, there must be a difference in the number and/or types and/or modifiers of the parameters.

Evaluation Copy

Method Overloading (Cont'd)

- **At runtime, the compiler will resolve a given invocation of the method by trying to match up the actual parameters with formal parameters.**
  - A match occurs if the parameters match exactly or if they can match through an implicit conversion.
  - For the exact matching rules, consult the **C# Language Specification**.
- **The program *OverloadDemo* illustrates method overloading.**
  - The method **FindMax** is overloaded to take either **long** or **string** parameters.
  - The method is invoked three times, for **int**, **long**, and **string** parameters.
  - There is an exact match for the case of **long** and **string**.
  - The call with **int** actual parameters can resolve to the **long** version, because there is an implicit conversion of **int** into **long**.
  - You may wish to review the discussion of conversions of data types at the end of Chapter 3.
- **We will cover the *string* data type and the *Compare* method in Chapter 10.**

Evaluation Copy

Lab 9A

Method Overloading

In this lab, you will extend the **SimpleMath** class to include subtraction and division, providing the four methods for **double** as well as **int**.

Detailed instructions are contained in the Lab 9A write-up at the end of the chapter.

Suggested time: 15 minutes

Evaluation Copy

Copyright © 2012 Object Innovations Enterprises, LLC
All Rights Reserved

Modifiers as Part of the Signature

- **It is important to understand that if methods have identical types for their formal parameters, but differ in a modifier (none, *ref*, or *out*), then the methods have different signatures.**
- **The program *OverloadHotel* provides an illustration.**
  - We have two **RaisePrice** methods.
  - In the first method, the hotel is passed as a value parameter.
  - In the second version, the hotel is passed as a reference parameter.
  - These methods have different signatures.

Evaluation Copy

Copyright © 2012 Object Innovations Enterprises, LLC
All Rights Reserved

Variable Length Parameter Lists

- **Our *FindMax* methods in the previous section were very specific with respect to the number of parameters—there were always exactly two parameters.**
- **Sometimes you may want to be able to work with a variable number of parameters, for example, to find the maximum of two, three, four, or more numbers.**
- **C# provides the *params* keyword, which you can use to indicate that an array of parameters is provided.**
  - Sometimes you may want to provide both a general version of your method that takes a variable number of parameters and also one or more special versions that take an exact number of parameters.
  - The special version will be called in preference, if there is an exact match. The special versions are more efficient.
- **The program *VariableMax* illustrates a general *FindMax* method that takes a variable number of parameters.**
  - There is also a special version that takes two parameters.
  - Each method prints out a line identifying itself, so you can see which method takes precedence.

Evaluation Copy

Properties

- **The encapsulation principle leads us to typically store data in private fields and to provide access to this data through public accessor methods that allow us to set and get values.**
  - For example, in the **Account** class we used as an illustration in Chapter 7, we provided a method **GetBalance** to access the private field **balance**.
  - You don’t need any special syntax; you can simply provide methods and call these methods what you want, typically **GetXXX** and **SetXXX**.
- **C# provides a special property syntax that simplifies user code.**
- **Rather than using methods, you can simply use an object reference, followed by a dot, followed by a property name.**
  - Some examples of a **Balance** property (that is both read/write) of a **SimpleAccount** class follow.
  - We show in comments the corresponding method code.

Evaluation Copy

Properties Examples

- **First example is *SimpleAccount*.**

```
SimpleAccount acc = new SimpleAccount();
decimal bal;
bal = acc.Balance;
// bal = acc.GetBalance();
acc.Balance = 100m;
acc.Balance += 1m;
// acc.SetBalance(acc.GetBalance() + 1m);
```

- **As you can see, the syntax using the property is a little more concise.**
- **Properties were popularized in Visual Basic and are now part of .NET and available in other .NET languages, such as C#.**
- **The program *AccountProperty* illustrates implementing and using several properties: *Balance*, *Id*, and *Owner*.**
  - The first two properties are read-only (only **get** defined) and the third property is read/write (both **get** and **set**).
  - It is also possible to have a write-only property (only **set** defined).
- **The next page shows the code for the *Account* class, where the properties are defined.**
  - Notice the syntax and the special C# keyword **value**.

Evaluation Copy

Properties Example (Cont’d)

```
// Account.cs

public class Account
{
   private int id;
   private static int nextid = 1;
   private decimal balance;
   private string owner;
   public Account(decimal balance, string owner)
   {
      this.id = nextid++;
      this.balance = balance;
      this.owner = owner;
   }
   public void Deposit(decimal amount)
   {
      balance += amount;
   }
   public void Withdraw(decimal amount)
   {
      balance -= amount;
   }
   public decimal Balance
   {
      get
      {
         return balance;
      }
   }
   public int Id
   {
      get
      {
         return id;
      }
   }
```

Evaluation Copy

Properties Example (Cont’d)

```
   public string Owner
   {
      get
      {
         return owner;
      }
      set
      {
         owner = value;
      }
   }
}
```

Evaluation Copy

Copyright © 2012 Object Innovations Enterprises, LLC
All Rights Reserved

Auto-Implemented Properties

- **An auto-implemented property provides a concise way of defining a property.**
  - The compiler automatically provides a private field to implement the property.
  - You can only access the property through the **get** and **set** accessors.
- **An auto-implemented property must declare both a *get* and a *set* accessor.**

```
public decimal Balance { get; set; }
```

- **An auto-implemented property can be made read-only by declaring *set* as private.**

```
public int AccountId { get; private set; }
// read-only
```

- **The next page provides an example of the use of auto-implemented properties.**
  - See **AutoProperties**.
  - The line that is commented out is illegal because the property **AccountId** is read-only.

Evaluation Copy

Auto-Implemented Property Example

```
class Program
{
   static void Main(string[] args)
   {
      Account acc = new Account(101, 150m);
      acc.Show();
      acc.Balance += 100m;
      acc.Show();
      // acc.AccountId = 201;
      // Illegal because AccountId is read-only
   }
}

class Account
{
   public int AccountId { get; private set; }
   // readonly
   public decimal Balance { get; set; }
   public Account(int accId, decimal bal)
   {
      AccountId = accId;
      Balance = bal;
   }
   public void Show()
   {
      Console.WriteLine("Id: {0}, Balance: {1:C}",
         AccountId, Balance);
   }
}
```

Evaluation Copy

Lab 9B

Properties

In this lab, you will use properties to access and modify member data items in an object of a class type.

Detailed instructions are contained in the Lab 9B write-up at the end of the chapter.

Suggested time: 20 minutes

Evaluation Copy

Copyright © 2012 Object Innovations Enterprises, LLC
All Rights Reserved

Operator Overloading

- **Another kind of syntactic simplification that can be provided in C# is *operator overloading*.**
- **The idea is that certain method invocations can be implemented more concisely using operators, rather than method calls.**
  - Suppose we have a class **Matrix** that has static methods to add and multiply matrices.
  - Using methods, we could write a matrix expression like this:

```
Matrix a, b, c, d;
// code to initialize the object references
d = Matrix.Multiply(a, (Matrix.Add(b, c));
```

  - If we overload the operators + and *, we can write this code more succinctly:

```
d = a * (b + c);
```

Evaluation Copy

Operator Overloading (Cont'd)

- **You cannot create a brand new operator, but you can overload many of the existing C# operators to be an alias for a static method.**
  - For example, given the static method **Add** in the **Matrix** class ...

```
class Matrix
{
...
   public static Matrix Add(Matrix x, Matrix y)
   {
```

  - ... you could write instead:

```
   public static Matrix operator+(Matrix x,
                                 Matrix y)
```

- **All of the rest of the class implementation code stays the same, and you can then use operator notation in client code. Operator declarations, such as *operator+* shown above, must obey the following rules:**
  - Operators must be **public** and **static**, and may not have any other modifiers.
  - Operators take only value parameters, and not reference or output parameters.
  - Operators must have a signature that differs from the signatures of all other operators in the class.

Evaluation Copy

Operator Overloading (Cont'd)

- **There are three categories of operators that can be overloaded.**
  - The table shows the unary and binary operators that can be overloaded.
  - A third category of operators, user-defined conversions, will be discussed in Chapter 14.

Type	Operators
Unary	+ - ! ~ ++ -- true false
Binary	+ - * / % & | ^ << >> == != > < >= <=

  - If you overload a binary operator **op**, the corresponding compound assignment, operator **op=,** will be overloaded for you by the compiler. For example, if you overload +, you will automatically have an overload of +=.

- **The relational operators must be overloaded in pairs:**
  - operator== and operator!=
  - operator> and operator<
  - operator>= and operator<=.

Evaluation Copy

Sample Program

- **As an illustration of operator overloading, consider the program *ClockOverload*, which has a class, *Clock*, that does “clock arithmetic.”**
  - The legal values of **Clock** are integers between 1 and 12 inclusive.
  - Addition is performed modulo 12. Thus 9 + 7 is 16 modulo 12, or 4.
  - We overload the plus operator to perform this special kind of addition operation.
  - We have two different versions of the plus operator. One adds two **Clock** values, and the other adds a **Clock** and an **int**.
  - In the test program, note that we are able to use +=, even though we have not explicitly provided such an overload. The compiler automatically furnishes this overload for us, by virtue of our overloading +.

Evaluation Copy

Operator Overloading in the Class Library

- **Although you may rarely have occasion to overload operators in your own classes, you will find that a number of classes in the .NET Framework Class Library make use of operator overloading.**
- **In Chapter 10, you will see how + is used for concatenation of strings.**
- **In Chapter 18, you will see how += is used for adding an event handler to an event.**

Evaluation Copy

Summary

- **In this chapter, we examined a number of features of methods.**
- **In C#, there is no such thing as a freestanding function.**
- **All functions are tied to classes and are called methods.**
- **If you do not care about class instances, you can implement a class that has only static methods.**
- **By default, parameters are passed by value, but C# also supports reference parameters and output parameters.**
- **A method name can be overloaded, with different versions having different parameter lists.**
- **You can also implement methods in C# that take a variable number of parameters.**
- **C# provides special property syntax for concisely invoking get/set methods for accessing data.**
- **You can overload operators in C#, a feature which makes the C# language inherently more extensible without requiring special coding in the compiler.**

Evaluation Copy

Lab 9A

Overloading Methods

Introduction

In this lab, you will extend the **SimpleMath** class to include subtraction and division, providing the four functions for **double** as well as **int**.

Suggested Time: 15 minutes

Root Directory: **OIC\CSharp**

Directories:	**Labs\Lab9A\TestSimpleMath**	(work area)
	Chap09\TestSimpleMath\Step1	(backup of starter files)
	Chap09\TestSimpleMath\Step2	(answer)

Instructions

1. Build and run the starter project.

2. Extend the **SimpleMath** class by adding functions to handle subtraction and division for **int**.

3. Then add methods to handle the same four operations for **double**.

4. Add test code to **TestSimpleMath.cs** to check your overloaded methods.

5. Build and test.

Evaluation Copy

Lab 9B

Properties

Introduction

In this lab, you will use properties to access and modify member data items in an object of a class type.

Suggested Time: 20 minutes

Root Directory: **OIC\CSharp**

Directories:	**Labs\Lab9B\HotelClass**	(work area)
	Chap09\HotelClass\Step1	(backup of starter files)
	Chap09\HotelClass\Step2	(answer)

Instructions

1. Build and run the starter project.

2. Change all of the data members of the **Hotel** class to have private access, and add properties to access and change the data members. In the properties for the number of rooms, enforce a limitation of no more than 400 and no fewer than 10. For the cost of a room, limit it to the range $30-$150.

3. Modify **HotelTest.cs** to check the properties.

4. Build and test.

Evaluation Copy

Evaluation Copy

Copyright © 2012 Object Innovations Enterprises, LLC
All Rights Reserved