

Object-Oriented Programming in C#

Student Guide

Revision 4.6

Object-Oriented Programming in C#

Rev. 4.6

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, Howard Lee Harkness

Special Thanks: Marianne Oberg, Sharman Staples, Paul Nahay

Copyright ©2013 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.

Table of Contents (Overview)

Chapter 1	.NET: What You Need To Know
Chapter 2	First C# Programs
Chapter 3	Data Types in C#
Chapter 4	Operators and Expressions
Chapter 5	Control Structures
Chapter 6	Object-Oriented Programming
Chapter 7	Classes
Chapter 8	More about Types
Chapter 9	Methods, Properties and Operators
Chapter 10	Characters and Strings
Chapter 11	Arrays and Indexers
Chapter 12	Inheritance
Chapter 13	Virtual Methods and Polymorphism
Chapter 14	Formatting and Conversion
Chapter 15	Exceptions
Chapter 16	Interfaces
Chapter 17	.NET Interfaces and Collections
Chapter 18	Delegates and Events
Chapter 19	Introduction to Windows Forms
Chapter 20	New Features in C# 4.0 and C# 5.0
Appendix A	Learning Resources

Electronic File Supplements

Supplement1.pdf	Using Visual Studio 2013
Supplement2.pdf	Language Integrated Query (LINQ)
Supplement3.pdf	Unsafe Code and Pointers in C#

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** and **Supp3**.

Table of Contents (Detailed)

Chapter 1 .NET: What You Need to Know	1
Getting Started	3
.NET: What Is <i>Really</i> 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 2013.....	10
Creating a Console Application.....	11
Adding a C# File.....	12
Using the Visual Studio Text Editor.....	13
IntelliSense	14
Build and Run the Project.....	15
Pausing the Output.....	16
Visual C# and GUI Programs	17
.NET Documentation	18
Summary	19
Chapter 2 First C# Programs	21
Hello, World	23
Compiling, Running (Command Line).....	24
Program Structure	25
Namespaces	28
Exercise.....	29
Answer.....	30
Variables	31
Expressions	32
Assignment	33
Calculations Using C#	34
Sample Program.....	35
More about Output in C#.....	36
Input in C#	37
More about Classes.....	38
InputWrapper Class	39
Echo Program	40
Using InputWrapper	41
Compiling Multiple Files.....	42
Multiple Files in Visual Studio.....	43
The .NET Framework.....	44
Lab 2	46
Summary	47

Chapter 3 Data Types in C#.....	51
Strong Typing	53
Typing in C#	54
Typing in C++.....	55
Typing in Visual Basic 6	56
C# Types	57
Integer Types	58
Integer Type Range.....	59
Integer Literals.....	60
Floating Point Types.....	61
Floating Point Literals	62
IEEE Standard for Floating Point.....	63
Decimal Type.....	64
Decimal Literals.....	65
Character Type.....	66
Character Literals.....	67
string	68
Escape Characters	69
Boolean Type.....	70
Implicit Conversions.....	71
Explicit Conversions.....	72
Conversions Example	73
Nullable Types.....	74
Lab 3	75
Summary	76
Chapter 4 Operators and Expressions.....	79
Operator Cardinality	81
Arithmetic Operators	82
Multiplication	83
Division.....	84
Additive Operators.....	85
Increment and Decrement.....	86
Example: A Small Calculator	87
Relational Operators	88
Conditional Logical Operators	89
Short-Circuit Evaluation.....	90
Ternary Conditional Operator.....	92
Bitwise Operators	93
Bitwise Logical Operators	94
Bitwise Shift Operators.....	95
Assignment Operators.....	96
Expressions	97
Precedence	98
Associativity	99
Checking.....	100

Lab 4	101
Summary	102
Chapter 5 Control Structures.....	105
If Test	107
Blocks	108
Loops	109
while Loop	110
do/while Loops	111
for Loops	112
ForUp Example	113
ForDown Example	114
Arrays.....	115
Fibonacci Example	116
foreach Loop	117
break.....	118
continue.....	119
goto	120
Structured Programming.....	121
Structured Search Example.....	122
Multiple Methods.....	123
switch	125
switch in C# and C/C++	126
Lab 5	127
Summary	128
Chapter 6 Object-Oriented Programming.....	131
Objects	133
Objects in the Real World.....	134
Object Models.....	135
Reusable Software Components	136
Objects in Software.....	137
State and Behavior	138
Abstraction.....	139
Encapsulation.....	140
Classes	141
Inheritance Concept	142
Inheritance Example	143
Relationships among Classes.....	144
Polymorphism.....	145
Object Oriented Analysis and Design.....	147
Use Cases.....	148
CRC Cards and UML	149
Summary	150
Chapter 7 Classes.....	151
Classes as Structured Data.....	153

Classes and Objects	154
References.....	155
Instantiating and Using an Object.....	156
Assigning Object References.....	157
Garbage Collection	158
Sample Program.....	159
Methods	160
Method Syntax Example.....	161
Public and Private	162
Abstraction.....	164
Encapsulation.....	165
Initialization	166
Initialization with Constructors	167
Default Constructor.....	169
this.....	170
TestAccount Sample Program	171
Static Fields and Methods.....	173
Static Methods	174
Sample Program.....	175
Static Constructor	176
Constant and Readonly Fields	177
Lab 7	178
Summary	179
Chapter 8 More about Types.....	183
Overview of Types in C#.....	185
Structures	186
Uninitialized Variables	187
Test Program.....	188
Copying a Structure	189
Hotel.cs	190
HotelCopy.cs.....	191
Results of Hotel Copy.....	192
Classes and Structs	193
Enumeration Types.....	194
Enumeration Types Examples	195
Reference Types	196
Class Types.....	197
object.....	198
string	199
Arrays.....	200
Default Values	201
Boxing and Unboxing.....	203
Implicitly Typed Variables.....	204
Implicitly Typed Variables – Example.....	205
Lab 8	206

Summary	207
Chapter 9 Methods, Properties, and Operators	209
Static and Instance Methods	211
Method Parameters	212
No “Freestanding” Functions in C#.....	213
Classes with All Static Methods	214
Parameter Passing	215
Parameter Terminology	216
Value Parameters	217
Reference Parameters	218
Output Parameters.....	221
Structure Parameters	222
Class Parameters	223
Method Overloading	224
Lab 9A	226
Modifiers as Part of the Signature	227
Variable Length Parameter Lists	228
Properties	229
Properties Examples	230
Auto-Implemented Properties.....	233
Auto-Implemented Property Example	234
Lab 9B.....	235
Operator Overloading	236
Sample Program.....	239
Operator Overloading in the Class Library	240
Summary	241
Chapter 10 Characters and Strings	245
Characters	247
Sample Program.....	248
Character Codes.....	249
ASCII and Unicode.....	250
Escape Sequences	251
Strings	252
String Class	253
String Literals and Initialization	254
Concatenation	255
Index	256
Relational Operators	257
String Equality	258
String Comparisons.....	259
String Comparison	260
String Input	262
String Methods and Properties.....	263
StringBuilder Class	265

StringBuilder Equality	267
Command Line Arguments.....	268
Command Line Arguments in the IDE.....	269
Command Loops.....	270
Splitting a String.....	271
Lab 10.....	272
Summary.....	273
Chapter 11 Arrays and Indexers.....	277
Arrays.....	279
One Dimensional Arrays	280
System.Array	281
Sample Program.....	282
Random Number Generation.....	283
Next Methods.....	284
Jagged Arrays	285
Rectangular Arrays.....	286
Arrays as Collections.....	287
Bank Case Study: Step 1.....	289
Account Class.....	291
Bank Class.....	294
TestBank Class.....	297
Atm Class.....	299
Running the Case Study.....	301
Indexers.....	303
ColorIndex Example Program	305
Using the Indexer.....	306
Lab 11.....	307
Summary.....	308
Chapter 12 Inheritance.....	311
Inheritance Fundamentals.....	313
Inheritance in C#.....	314
Single Inheritance.....	315
Root Class – <i>Object</i>	316
Access Control.....	317
Public Class Accessibility.....	318
Internal Class Accessibility	319
Member Accessibility.....	320
Member Accessibility Qualifiers.....	321
Member Accessibility Example.....	322
Method Hiding.....	323
Method Hiding and Overriding	324
Example: Method Hiding.....	325
Initialization.....	326
Initialization Fundamentals	327

Initialization Fundamentals Example	328
Default Constructor.....	329
Overloaded Constructors	330
Example: Overloaded Constructors	331
Invoking Base Class Constructors	332
Base Class Initialization Example	333
Bank Case Study: Step 2.....	334
Bank Case Study Analysis.....	335
Account.....	336
CheckingAccount.....	337
SavingsAccount	339
TestAccount.....	341
Running the Case Study.....	342
Lab 12	343
Summary	344
Chapter 13 Virtual Methods and Polymorphism	347
Introduction to Polymorphism.....	349
Abstract and Sealed Classes	350
Virtual Methods and Dynamic Binding.....	351
Type Conversions in Inheritance.....	352
Converting Down the Hierarchy.....	353
Converting Up the Hierarchy.....	354
Virtual Methods	355
Virtual Method Example	356
Virtual Method Cost	357
Method Overriding	358
The Fragile Base Class Problem.....	359
<i>override</i> Keyword.....	360
Polymorphism.....	361
Polymorphism Using “Type Tags”	362
Polymorphism Using Virtual	363
Polymorphism Example.....	364
Abstract Classes.....	368
Keyword: <i>abstract</i>	369
Sealed Classes.....	370
Heterogeneous Collections	371
Heterogeneous Collections Example.....	372
Bank Case Study: Step 3.....	373
Case Study Classes	374
Run the Case Study.....	376
Account.....	377
CheckingAccount, SavingsAccount	378
Bank and Atm	379
TestBank	380
Lab 13	381

Summary	382
Chapter 14 Formatting and Conversion.....	385
Introduction to Formatting.....	387
ToString	388
ToString in Your Own Class	389
Using Placeholders	391
Format Strings.....	392
Simple Placeholders.....	393
Controlling Width.....	394
Format String	395
Currency.....	396
Currency Format Example.....	397
String.Format	398
PadLeft and PadRight	399
Bank Case Study: Step 4.....	401
Type Conversions	402
Conversion of Built-In Types	403
Conversion of User-Defined Types	404
User Defined Conversions: Example.....	406
Lab 14	408
Summary	409
Chapter 15 Exceptions.....	411
Introduction to Exceptions.....	413
Exception Fundamentals.....	414
.NET Exception Handling.....	415
Exception Flow of Control	416
Context and Stack Unwinding	417
Exception Example	418
System.Exception	421
User-Defined Exception Classes	422
User Exception Example	423
Structured Exception Handling.....	426
Finally Block.....	427
Bank Case Study: Step 5.....	429
Inner Exceptions	430
Checked Integer Arithmetic.....	431
Example Program	432
Lab 15	433
Summary	434
Chapter 16 Interfaces	437
Introduction.....	439
Interfaces in C#.....	441
Interface Inheritance	442
Programming with Interfaces.....	443

Implementing Interfaces	444
Using an Interface	446
Demo: SmallInterface	447
Dynamic Use of Interfaces	448
Demo: TryInterfaces	449
is Operator.....	450
as Operator.....	451
Bank Case Study: Step 6.....	452
Common Interfaces in Case Study –IAccount.....	453
Apparent Redundancy.....	454
IStatement	455
IStatement Methods	456
IChecking.....	457
ISavings	458
The Implementation.....	459
SavingsAccount	460
The Client	461
Resolving Ambiguity.....	463
Access Modifier.....	464
Explicit Interfaces Test Program.....	465
Summary	466
Chapter 17 .NET Interfaces and Collections.....	467
Overview.....	469
Collections	470
ArrayList Example.....	471
Count and Capacity.....	472
foreach Loop.....	473
Array Notation	474
Adding to the List	475
Remove Method.....	476
RemoveAt Method.....	477
Collection Interfaces	478
IEnumerable and IEnumerator.....	479
IEnumerable and IEnumerator Demo: <i>AccountList</i>	480
ICollection	481
IList.....	482
A Collection of User-Defined Objects.....	483
Duplicate Objects.....	484
A Correction to AccountList (Step 1).....	485
Bank Case Study: Step 7.....	486
Copy Semantics and ICloneable	487
Copy Semantics in C#.....	488
Shallow Copy and Deep Copy	489
Example Program	490
Reference Copy.....	491

Memberwise Clone	492
Using ICloneable	493
Comparing Objects	494
Sorting an Array.....	495
Anatomy of Array.Sort	496
Using the is Operator	497
The Use of Dynamic Type Checking	498
Implementing IComparable	499
Running the Program.....	500
Complete Solution	501
Lab 17A	502
Writing Generic Code.....	503
Using a Class of <i>object</i>	504
Generic Types.....	505
Generic Syntax in C#.....	506
Generic Example.....	507
Generic Client Code.....	508
System.Collections.Generic.....	509
Lab 17B.....	510
Object Initializers.....	511
Collection Initializers.....	512
Anonymous Types	513
Summary	514
Chapter 18 Delegates and Events.....	519
Overview of Delegates and Events.....	521
Callbacks and Delegates	522
Usage of Delegates	523
Declaring a Delegate.....	524
Defining a Method.....	525
Creating a Delegate Object.....	526
Calling a Delegate.....	527
A Random Array.....	528
Anonymous Methods.....	529
Combining Delegate Objects	530
Account.cs.....	531
DelegateAccount.cs	532
Lambda Expressions	533
Named Method	534
Anonymous Method	535
Lambda Expression Example	536
Events.....	537
Events in C# and .NET	538
Client Side Event Code.....	540
Chat Room Example.....	541
Lab 18	542

Summary	543
Chapter 19 Introduction to Windows Forms	545
Creating a Windows Forms App	547
Partial Classes	551
Windows Forms Event Handling.....	552
Add Events for a Control	553
Events Documentation	554
Closing a Form.....	555
ListBox Control	556
ListBox Example	557
Lab 19	558
Summary	559
Chapter 20 New Features in C# 4.0 and C# 5.0	563
<i>dynamic</i> Type.....	565
Runtime Error Example	566
<i>dynamic</i> versus <i>object</i>	567
Behavior of <i>object</i>	568
Behavior of <i>dynamic</i>	569
Named Arguments	570
Optional Arguments.....	571
Book Class	572
Using Optional Arguments	573
Variance in Generic Interfaces	574
Covariance Example	575
Variance with <code>IComparer<T></code>	576
Interfaces with Variance Support	577
Contravariance Example.....	578
Asynchronous Programs in C# 5.0	579
Task and <code>Task<TResult></code>	580
Aysnc Methods	581
Async Example	582
Synchronous Call.....	583
Async Call.....	584
Threading	585
Summary	586
Appendix A Learning Resources	587

EVALUATION COPY
Unauthorized reproduction or distribution is prohibited.

Chapter 1

.NET: What You Need to Know

EVALUATION COPY
Unauthorized reproduction or distribution is prohibited.

.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 2013 as an effective environment for creating C# 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 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.

.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 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.

.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 2013 | Visual Studio Tools | Developer Command Prompt for VS2013. Navigate to folder \OIC\CsEss\Chap01\SimpleCalc.

```
>csc SimpleCalc.cs
```

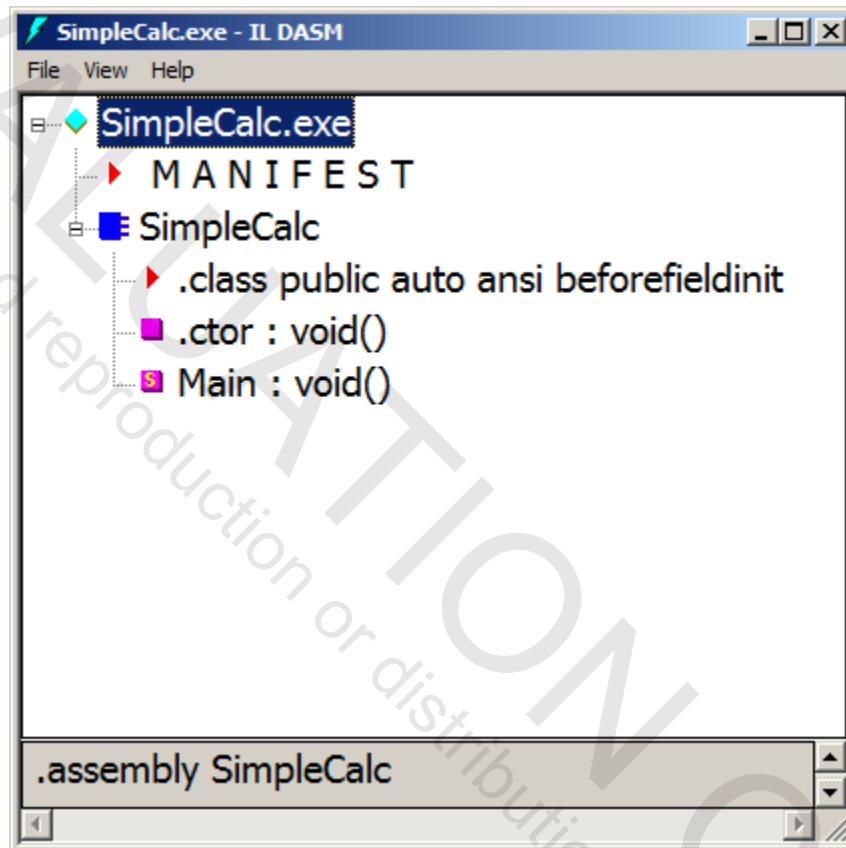
3. Run your IL program **SimpleCalc.exe**

```
>SimpleCalc
area = 100
```

Viewing the Assembly

- You can view the assembly using the *ILDASM* tool¹.

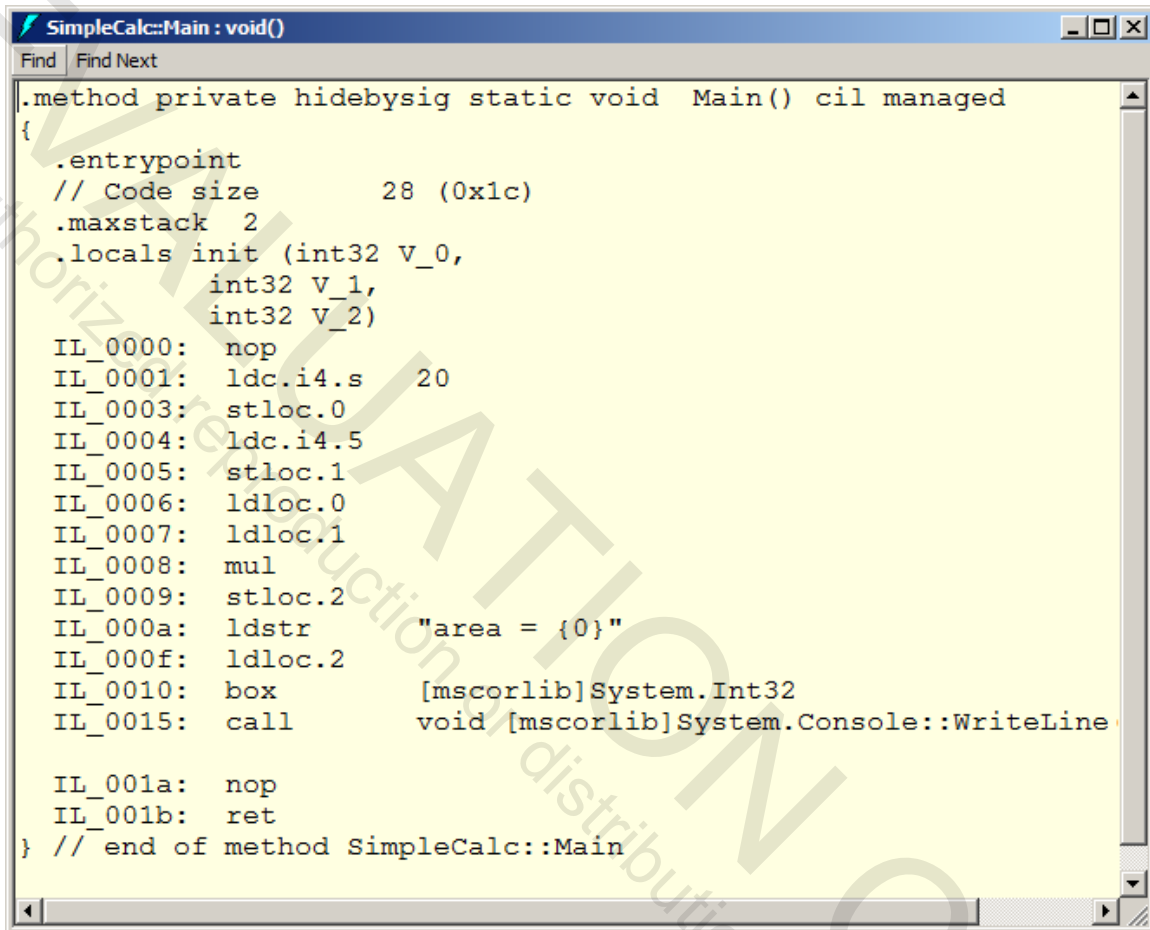
```
>ildasm SimpleCalc.exe
```



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

Viewing Intermediate Language

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



```
SimpleCalc::Main : void()
Find Find Next
.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
```


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

ADO.NET

XML Programming

Windows

Windows Forms

WPF

Web

ASP.NET

ASP.NET MVC

WCF

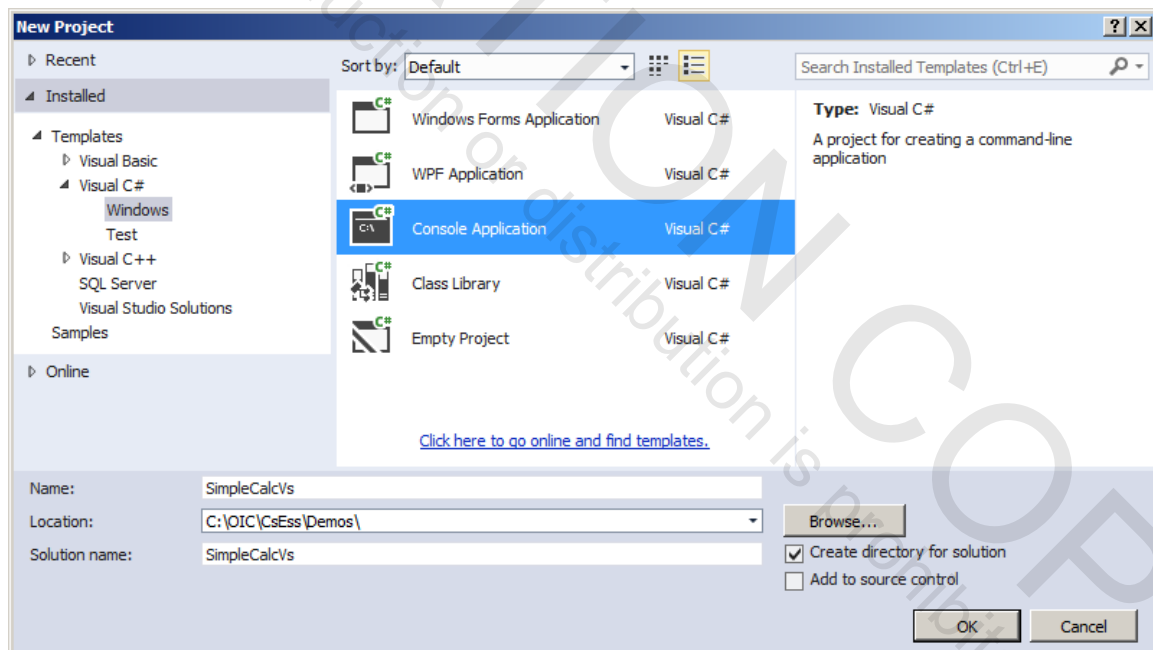
ASP.NET AJAX

Visual Studio 2013

- **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 2013 throughout this course to create and compile your C# programs.**
- **Visual Studio 2013 is discussed in more detail in Appendix A.**
- **In this course we use the free version VS Express 2013 for Desktop.**

Creating a Console Application

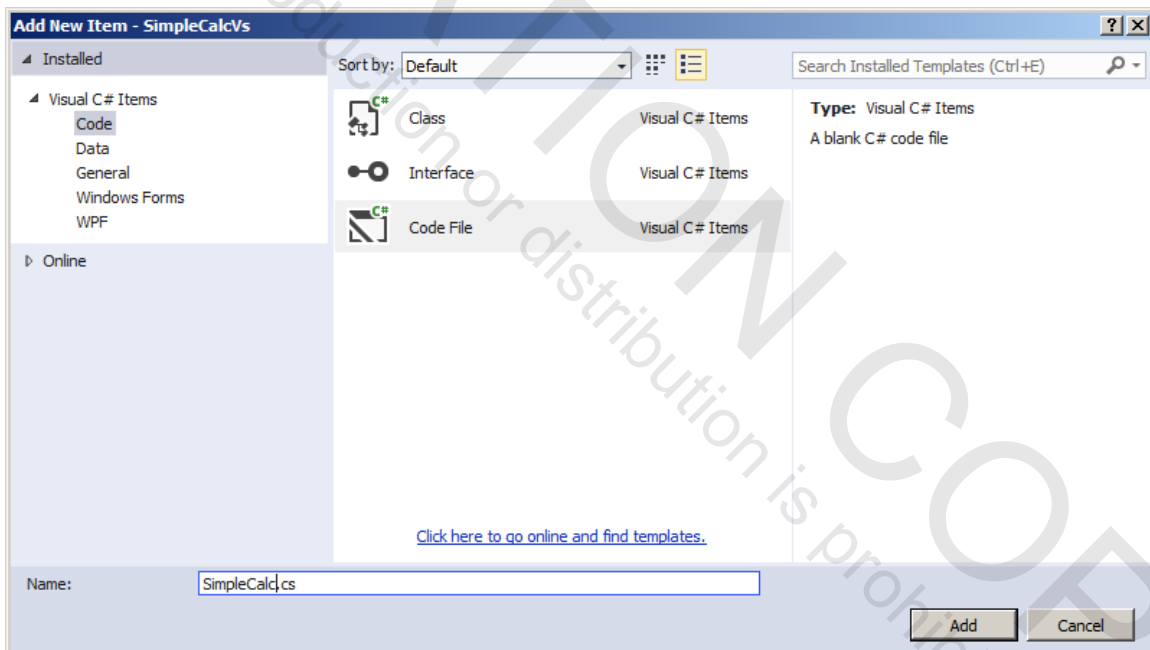
- **We will now create a simple console application.**
 - Our program is the same simple calculator we created earlier that was compiled at the command line.
- 1. From the Visual Studio main menu, choose File | New | Project.... This will bring up the New Project dialog.
- 2. Choose Visual C# and “Console Application”.
- 3. Leave checked “Create directory for solution”.²
- 4. In the Name field, type **SimpleCalcVs** and for Location browse to **C:\OIC\CsEss\Demos**. Click OK.



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

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.



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**.

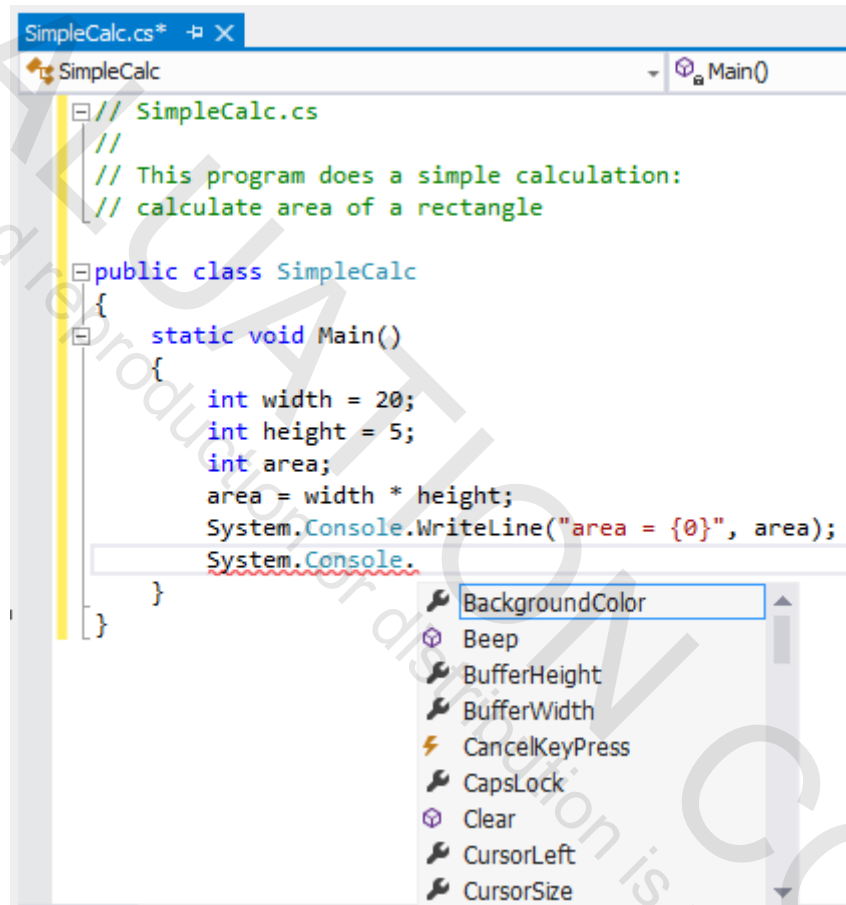
```
// SimpleCalc.cs
//
// This program does a simple calculation:
// calculate area of a rectangle

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.

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.



The screenshot shows a Visual Studio code editor window titled 'SimpleCalc.cs*'. The code is as follows:




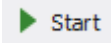
```
// SimpleCalc.cs
//
// This program does a simple calculation:
// calculate area of a rectangle

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.
    }
}
```

An IntelliSense list box is open below the cursor, showing a list of members from the `System.Console` class. The members listed are:

- BackgroundColor
- Beep
- BufferHeight
- BufferWidth
- CancelKeyPress
- CapsLock
- Clear
- CursorLeft
- CursorSize

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.

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.**

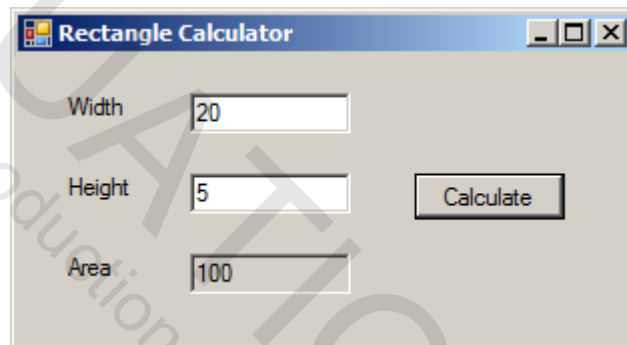
```
public class SimpleCalc
{
    static void Main()
    {
        int width = 20;
        int height = 5;
        int area;
        area = width * height;
        System.Console.WriteLine("area = {0}", area);
        System.Console.WriteLine(
            "Prese Enter to exit");
        System.Console.ReadLine();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

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

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#.

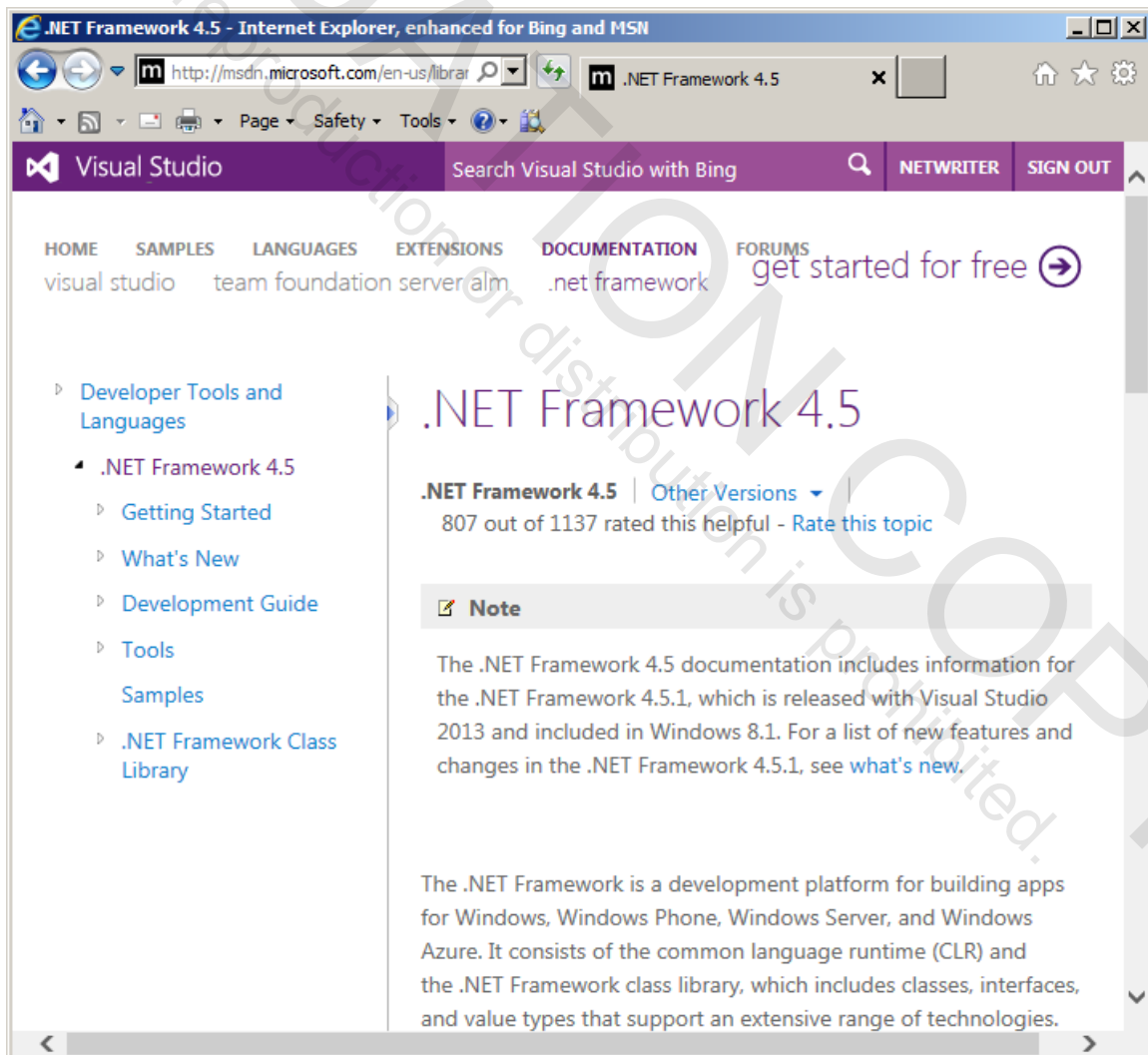
Example: See `Chap01\SimpleCalcGui`



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

.NET Documentation

- **.NET Framework documentation is included with Visual Studio 2013.**
 - 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.
 - The .NET Framework 4.5 documentation now includes documentation for .NET 4.5.1, which is the version of the .NET Framework that comes with Visual Studio 2013.



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 2013 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
Unauthorized reproduction or distribution is prohibited.

Chapter 2

First C# Programs

EV
Unauthor
reproduct
or distrib
is prohibi
COPY

First C# Programs

Objectives

After completing this unit you will be able to:

- Write a basic “Hello, World” program in C#.
- Compile and run C# programs in your local development environment.
- Describe the basic structure of C# programs.
- Describe how related C# classes can be grouped into namespaces.
- Use variables and simple expressions in C# programs.
- Write C# programs that can perform simple calculations.
- Perform simple input and output in C#.
- Describe objects and classes in C#.
- Use an input wrapper class to perform input in C#.

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 C#:**
 - See **Demos\Hello\Hello.cs**, backed up in **Chap02\Hello**.

```
// Hello.cs

class Hello
{
    public static int Main(string[] args)
    {
        System.Console.WriteLine(
            "Hello, World");
        return 0;
    }
}
```

Compiling, Running (Command Line)

- **The Visual Studio 2013 IDE (integrated development environment) was introduced in Chapter 1, and we will use it throughout the course.**

- See Supplement 1 for more details.

- To open an existing project or solution, use the menu File | Open | Project/Solution. You can then navigate to a **.csproj** or **.sln** file.

- **If you are using the .NET SDK, you may do the following:**

- Compile the program via the command line:

```
csc Hello.cs
```

- 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.cs  
  
class Hello  
{  
    ...  
}
```

- **Every C# program has at least one *class*.**
 - A class is the foundation of C#'s support for object-oriented programming.
 - A class encapsulates data (represented by **variables**) and behavior (represented by **methods**).
 - All of the code defining the class (its variables and methods) will be contained between the curly braces.
 - We will discuss classes in detail later.
- **Note the *comment* at the beginning of the program.**
 - A line beginning with a double slash is present only for documentation purposes and is ignored by the compiler.
- **C# files have the extension *.cs*.**

Program Structure (Cont'd)

```
// Hello.cs
class Hello
{
    public static int Main(string[] args)
    {
        ...
        return 0;
    }
}
```

- **Every C# program has a distinguished class that has a method whose name must be *Main*.**
 - Note the capitalization!
 - The method should be **public** and **static**.
 - An **int** exit code can be returned to the operating system. Use **void** if you do not return an exit code.

```
public static void Main(string[] args)
```

- Command line arguments are passed as an array of strings.
- You may omit the command line arguments.

```
public static void Main()
```

- The runtime will call this **Main** method – it is the entry point for the program.
- All of the code for the **Main** method will be between the curly braces.

Program Structure (Cont'd)

```
// Hello.cs
class Hello
{
    public static int Main(string[] args)
    {
        System.Console.WriteLine(
            "Hello, World");
        return 0;
    }
}
```

- **Every method in C# has one or more *statements*.**
- **A statement is terminated by a semicolon.**
 - A statement may be spread out over several lines.
- **The *Console* class provides support for standard output and standard input.**
 - The method `WriteLine` displays a string, followed by a new line.

Namespaces

- **Much of the standard functionality in C# 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 the class name.**

```
System.Console
```

- **A *using* statement allows a class to be referred to by its class name alone.**

```
// Hello2.cs

using System;

class Hello
{
    public static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

- **Note that in C# it is not necessary for the file name to be the same as the name of the class containing the *Main* method.**
- **This version of the program also illustrates a *Main()* method with no command-line arguments and void return type.**

Exercise

- **Take a few minutes to add two more lines of code to your program.**
 - Print out the phrase “My name is XXXX” (use your own name).
 - Then print out “Goodbye.”
- **Save your file, compile the program, and run it.**
 - Your output should be something like this:

```
Hello, World  
My name is Bob  
Goodbye
```

Answer

```
// Hello3.cs
using System;
class Hello
{
    public static void Main()
    {
        Console.WriteLine("Hello, World");
        Console.WriteLine("My name is Bob");
        Console.WriteLine("Goodbye");
    }
}
```

Variables

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

```
int kilo = 1024;    // reserves space and assigns
                   // an initial value
int mega;          // reserves space but does
                   // not initialize
```

- If a variable is not initialized in its declaration, it should be assigned prior to being used.

```
int kilo;
kilo = 1024;
// Now you may use kilo
```

Expressions

- You can combine variables and constants (or “literals”) via *operators* to form *expressions*.
- Examples of operators include the standard arithmetic operators:

+	addition
-	subtraction
*	multiplication
/	division

- Here are some examples of expressions:

```
kilo * 1024
(fahrenheit - 32) * 5 / 9
3.1416 * radius * radius
```


Assignment

- **You can assign a value to a variable by using the = symbol.**
 - On the left hand side is a variable.
 - On the right hand side is an expression.
 - The expression is evaluated and its value is assigned to the variable on the left.
 - Assignment is a statement and must be terminated by a semicolon.

```
mega = kilo * 1024;  
celsius = (fahrenheit - 32) * 5 / 9;  
area = 3.1416 * radius * radius;
```

- **Note that the same variable can be used on both sides of an assignment statement.**

```
int item = 5;  
int total = 30;  
total = total + item;
```

- The expression **total + item** evaluates to 35, using the old value of **total**, and this value is assigned to **total**, creating a new value.

Calculations Using C#

- **You can easily use C# to perform calculations by adding code to the *Main* method of a C# class.**
 - Declare whatever variables you need.
 - Create expressions and assign values to your variables.
 - Print out the answer using **Console.WriteLine()**.
- **You can easily perform labeled output, relying on two features of C#:**
 - The operator + performs concatenation for **string** data.
 - There is an automatic, implicit conversion available that converts numeric data to string data when required.
 - Hence this code ...

```
int total = 35;  
System.Console.WriteLine("The total is " + total);
```

- ... will produce this output:

```
The total is 35
```

Sample Program

- **This program will convert temperature from Fahrenheit to Celsius.**

– See **Convert\Step1**.

```
// Convert.cs - Step 1
//
// Program converts a hardcoded temperature in
// Fahrenheit to Celsius

using System;

class Convert
{
    public static void Main(string[] args)
    {
        int fahr = 86;
        int celsius = (fahr - 32) * 5 / 9;
        Console.WriteLine("fahrenheit = " + fahr);
        Console.WriteLine("celsius = " + celsius);
    }
}
```

More about Output in C#

- **The *Console* class in the *System* namespace supports two simple methods for performing output:**

- **WriteLine()** writes out a string followed by a new line.
- **Write()** writes out just the string without the new line.

```
int x = 24;
int y = 5;
int z = x * y;
Console.Write("Product of " + x + " and " + y);
Console.WriteLine(" is " + z);
Console.WriteLine("The product is {0}", z);
```

- The output is all on one line:

```
Product of 24 and 5 is 120
```

- **A more convenient way to build up an output string is to use *placeholders* {0}, {1}, etc.**

- An equivalent way to do the output shown above is:

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

- The program **OutputDemo** illustrates the output operations just discussed.
- Later in the course we will see how to control formatting of output, and occasionally in examples we will throw in some simple use of formatting.

Input in C#

- **Our first Convert program is not too useful, because the Fahrenheit temperature is hard-coded.**
 - To convert a different temperature, you would have to edit the source file and recompile.
- **What we really want to do is allow the user of the program to enter a value at runtime for the Fahrenheit temperature.**
- **Although simple console input in C# is fairly easy, we can make it even easier using object-oriented programming.**
 - We can encapsulate or “wrap” the details of input in a class.
 - It will be easy to use the wrapper class.

More about Classes

- **Although we will discuss classes in more detail later, there is a little more you need to know now.**
- **A class can be thought of as a template for creating objects.**
 - An **object** is an instance of a **class**.
- **A class specifies data and behavior.**
 - The data is different for each object instance.
- **In C#, you instantiate a class by using the *new* keyword.**

```
InputWrapper iw = new InputWrapper();
```

- This code creates the object instance **iw** of the **InputWrapper** class.

InputWrapper Class

- **The *InputWrapper* class “wraps” interactive input for several basic data types.**
 - The supported data types are **int**, **double**, **decimal** and **string**.
 - Methods **getInt**, **getDouble**, **getDecimal** and **getString** are provided.
 - A prompt string is passed as an input parameter.
 - See files **InputWrapper.cs** in directory **TestInputWrapper**, which implements the class, and **TestInputWrapper.cs**, which tests the class.
- **Although the code is quite short, it is a little complex, involving a number of different methods from different .NET Framework classes.**
- **However, you do not need to be familiar with the implementation of *InputWrapper* in order to use it.**
 - That is the beauty of “encapsulation”—complex functionality can be hidden by an easy-to-use interface.

Echo Program

- **We illustrate interactive input by a simple “echo” program.**
 - The program prompts the user for a name, and then prints out a personalized greeting.
 - See **Echo**.
- **This directory has two files, each defining a class.**
 - **InputWrapper.cs** defines the wrapper class. There is no **Main** method in this class.
 - **EchoName.cs** has a class **Echo**, with a **Main** method.

```
// EchoName.cs
//
// Prompts user to enter name and then
// prints out greeting using name

using System;

class Echo
{
    public static void Main(string[] args)
    {
        InputWrapper iw = new InputWrapper();
        string name = iw.getString("Enter your name: ");
        Console.WriteLine("Hello, " + name);
    }
}
```


Using InputWrapper

- **The bolded statements illustrate how to use the *InputWrapper* class.**
 - Instantiate an **InputWrapper** object **iw** by using **new**.
 - Prompt to obtain input data by calling the appropriate **getXXX** method.

Compiling Multiple Files

- **It is easy to compile multiple files at the command line.**

```
csc *.cs
```

- This will compile all of the files in the current directory.
- The file containing a class with the **Main** method will be used as the name of the generated .EXE file:

```
Directory of C:\OIC\CSharp\Chap02\Echo
```

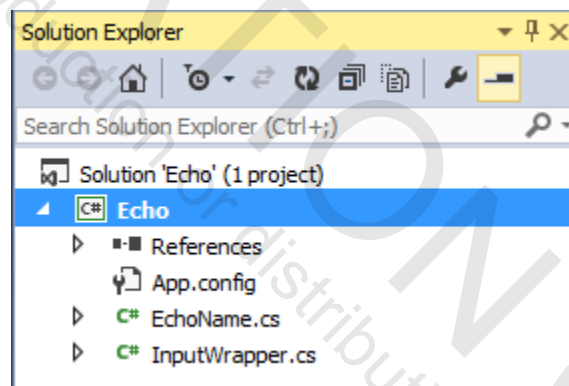
```
11/05/2013  08:01 PM    <DIR>          .
11/05/2013  08:01 PM    <DIR>          ..
11/05/2013  07:56 PM             187 App.config
11/05/2013  07:56 PM            2,088 Echo.csproj
11/05/2013  07:56 PM            1,004 Echo.sln
11/16/2001  07:21 PM             317 EchoName.cs
11/05/2013  07:59 PM            4,096 EchoName.exe
01/05/2001  01:34 PM             855 InputWrapper.cs
11/05/2013  07:56 PM           11,776 Echo.v12.suo
```



- If multiple classes contain a **Main** method, you can use the **/main** command line option to specify which class contains the **Main** method that you want to use as the entry point into the program.

```
csc *.cs /main:Echo
```

Multiple Files in Visual Studio

- It is very easy to work with Visual Studio projects that have multiple files.
- As an example, open the Visual Studio *solution* that is specified by the file *Chap02\Echo\Echo.sln*.
- A solution can contain one or more *projects*.
 - In this case there is a single project file **Echo.csproj**.
- You can see all the files in a solution through the *Solution Explorer*.



- When you build the solution via the toolbar button , you will build all the projects in the solution.
 - You can also build just the current project via the toolbar button .

The .NET Framework

- **The .NET Framework has a very large class library (several thousand classes).**
- **To make all of this functionality more manageable, the classes are partitioned into *namespaces*.**
- **The root namespace is *System*, which directly contains many useful classes, among them:**
 - **Console** provides access to standard input, output and error streams for I/O.
 - **Convert** provides conversions among base data types.
 - **Math** provides mathematical constants and functions.

The .NET Framework (Cont'd)

- **Underneath *System*, there are other namespaces, among them:**
 - **System.Data** contains classes constituting the ADO.NET architecture for accessing databases.
 - **System.Xml** provides standards-based support for processing XML.
 - **System.Drawing** contains classes providing GDI+ graphics functionality.
 - **System.Windows.Forms** provides support for creating applications with rich Windows-based interfaces.
 - **System.Web** provides support for browser/server communication.
 - **System.IO** provides support for reading and writing with streams and files. Both synchronous and asynchronous I/O are supported.
 - **System.Net** provides support for several standard network protocols.

Lab 2

C# Programs for Calculation

In this lab you modify or implement several C# programs to perform calculations. You need to perform input (through a wrapper class), perform a calculation, and output the result. Do as many of these exercises as time permits. If you have extra time, do some of the optional experiments suggested in some of the exercises, or make up some experiments on your own.

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

Suggested time: 30 minutes

Summary

- **Every C# application has a class with a method *Main*, which is the entry point into the application.**
- **The *System* class includes methods for performing output, such as *WriteLine*.**
- **Expressions in C# are formed from literals, variables and operators.**
- **With the assignment statement, you can assign a value computed by an expression to a variable.**
- **Input in C# is a little more complicated than output, but you can use a wrapper class that encapsulates the required C# classes and presents a simple programming interface.**
- **The .NET Framework has a large class library that is partitioned into namespaces.**

Lab 2

C# Programs for Calculation

Introduction

In this lab, you modify or implement several C# programs to perform calculations. You need to perform input (through a wrapper class), perform a calculation, and output the result. Do as many of these exercises as time permits. If you have extra time, do some of the optional experiments suggested in some of the exercises, or make up some experiments on your own.

Suggested Time: 30 minutes

Root Directory: OIC\CSharp

Directories: Labs\Lab2\Convert	(Exercise 1 work)
Chap02\Convert\Step1	(Backup of Exercise 1 starter files)
Chap02\Convert\Step2	(Answer to Exercise 1)
Chap02\TestInputWrapper	(InputWrapper class)
Labs\Lab2	(Exercise 2 work)
Chap02\Circle	(Exercise 2 answer)

Exercise 1. Fahrenheit to Celsius Conversion

Examine the code of the starter program. Build and run. Notice that the Fahrenheit temperature to be converted is hard-coded. Modify the program to prompt the user for a Fahrenheit temperature, read in the value entered by the user, and print out the result. Make use of the wrapper class **InputWrapper** that was discussed in this chapter.

The starter program uses **int** as the data type for temperatures. An optional experiment is to use **double** as the data type. Could you input the Fahrenheit temperature as an **int** and calculate the Celsius temperature as a **double**?

Exercise 2. Calculate the Area of a Circle

Use Visual Studio to create an empty C# project **Circle** in the **Lab2** folder. This will create the folder **Circle**. Add a new file **Circle.cs** to your project, where you will place your program code.

Write a C# program to calculate the area of a circle. Prompt the user for the radius, read in the value entered by the user, calculate the area of the circle, and print out the result. For pi, use the approximation 3.1416. What is an appropriate data type to use for radius and area?

As an optional experiment, use the class **Math** (in the namespace **System**) for a more accurate value of pi.

Another optional experiment is to capture the output data in a file. This is easy to do at the command line, by using the > to “redirect” output to a file.

```
Circle > output.txt
```

When using this technique, the prompts are written to the output file and not displayed on the screen. Hence you need to know what to type! If you type “10” for the radius, your output file should look like:

```
radius: Using, 3.1416, area = 314.16  
Using, Math.PI, area = 314.15926535897933
```

Chapter 5

Control Structures

EVALUATION COPY
Unauthorized reproduction or distribution is prohibited.

Control Structures

Objectives

After completing this unit you will be able to:

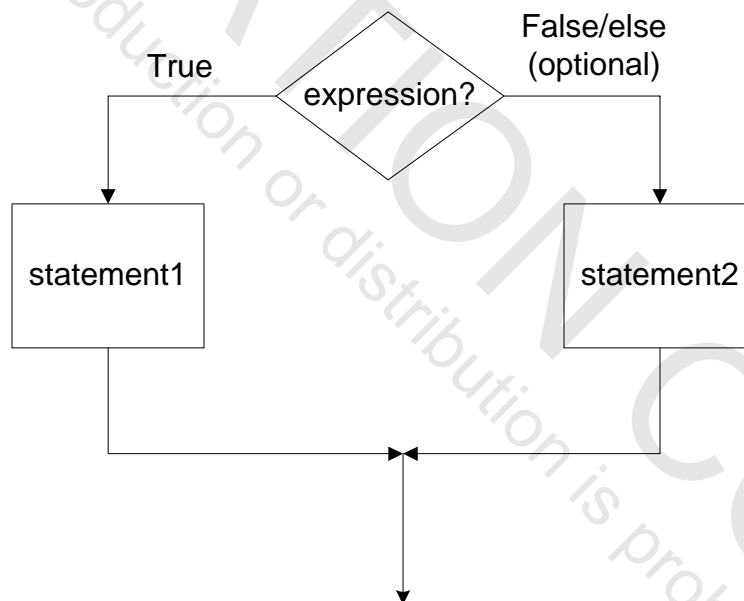
- **Use the common C# control structures to perform tests and loops.**
- **Use arrays in C# programs.**

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**.

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;
    }
}
```

Loops

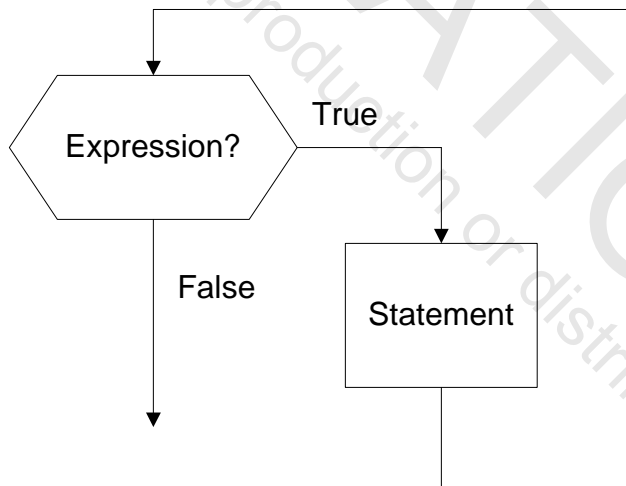
- **while**
- **for**
- **do/while**
- **foreach**
- **break**
- **continue**
- **goto**
- **switch**

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.

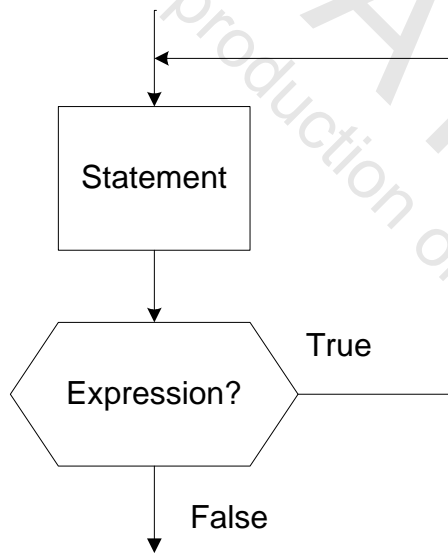


- See **LeapYearLoop**.

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!
```

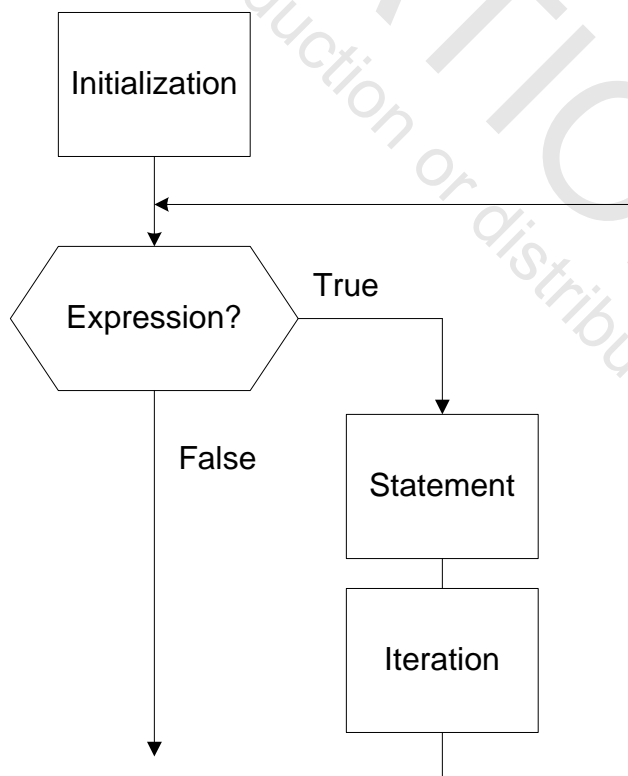


for Loops

- **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.



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;
    }
}
```

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;
    }
}
```

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};
```

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
```

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.WriteLine("{0} ", prime);
            sum += prime;
        }
        Console.WriteLine();
        Console.WriteLine("sum = {0}", sum);
        return 0;
    }
}
```

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

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.WriteLine("{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
```

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
```


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;
    }
}
```

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.

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.WriteLine("{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
```

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: ");
        }
    }
}
```

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
```

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**

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.**

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

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.**

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

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.

Chapter 9

Methods, Properties, and Operators

EVALUATION COPY
Unauthorized reproduction or distribution is prohibited.

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.**

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);
```

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.

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.

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;
    }
}
```

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.

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
    ...
}
```

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.

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.

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;
    }
}
```

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.

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
```


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**.

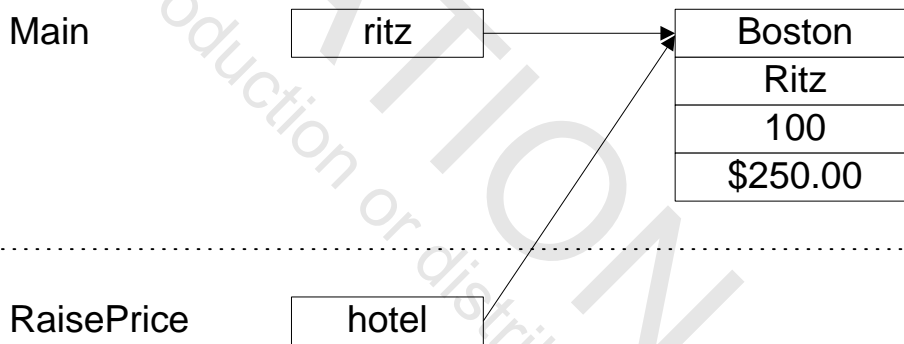
Main	ritz	Boston
		Ritz
		100
		\$200.00

RaisePrice	hotel	Boston
		Ritz
		100
		\$250.00

- 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.

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.

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.

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.**

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

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.

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.

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.

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**.

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;
        }
    }
}
```

Properties Example (Cont'd)

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

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.

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);
    }
}
```

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

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);
```

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.

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<=.

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 $+$.

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.**

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.**

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.

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.