

Object Oriented Programming in C#

**Robert J. Oberg
Howard Lee Harkness**

Student Guide
Revision 1.2

Object Oriented Programming in C#

Rev. 1.2

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.

Copyright ©2002 Object Innovations, Inc. All rights reserved.

Object Innovations, Inc.
4515 Emory Lane
Charlotte, NC 28211
704-362-5413
www.ObjectInnovations.com

Printed in the United States of America.

Table of Contents (Overview)

Chapter 1	.NET Framework
Chapter 2	First C# Programs
Chapter 3	Visual Studio .NET
Chapter 4	Simple Data Types
Chapter 5	Operators and Expressions
Chapter 6	Control Structures
Chapter 7	Object-Oriented Programming
Chapter 8	Classes
Chapter 9	The C# Type System
Chapter 10	Methods, Properties and Operators
Chapter 11	Characters and Strings
Chapter 12	Arrays and Indexers
Chapter 13	Inheritance
Chapter 14	Virtual Methods and Polymorphism
Chapter 15	Formatting and Conversion
Chapter 16	Exceptions
Chapter 17	Interfaces
Chapter 18	Interfaces and the .NET Framework
Chapter 19	Delegates and Events
Chapter 20	Advanced Features
Appendix A	Learning Resources

Table of Contents (Detailed)

Chapter 1 .NET Framework.....	1
.NET: What You Need To Know	3
.NET: What Is <i>Really</i> Happening	4
.NET Programming in a Nutshell	5
Understanding .NET	6
What Is .NET?	7
A New Programming Platform	8
Multiple Language Development.....	9
.NET Framework Overview.....	10
Common Language Runtime	11
.NET Framework Class Library.....	12
Common Language Specification	13
Languages in .NET.....	14
.NET Framework SDK	15
.NET Framework Class Library.....	16
Development Tools.....	17
Common Language Runtime	18
Design Goals of the CLR	19
Why Use a CLR?	20
Intermediate Language.....	21
Microsoft Intermediate Language	22
Metadata.....	23
JIT Compilation	24
Assemblies	25
Assembly Hierarchy.....	26
Types	27
Common Type System.....	28
Summary	29
Chapter 2 First C# Programs.....	31
Hello, World	33
Compiling, Running (Command Line)	34
Program Structure	35
Namespaces.....	38
Exercise.....	39
Answer	40
Variables	41
Expressions	42
Assignment	43
Calculations Using C#	44
Sample Program.....	45
Input in C#	46

More About Classes	47
InputWrapper Class	48
Echo Program	49
Using InputWrapper	50
Compiling Multiple Files	51
The .NET Framework	52
Lab 2	54
Summary	55
Chapter 3 Using Visual Studio.NET	61
Visual Studio.NET	63
Overview of VS.NET	64
Toolbars	67
Customizing a Toolbar	68
Creating a Console Application	69
Adding a C# File	70
Using the Visual Studio Text Editor	71
Build and Run the Bytes Project	72
Project Configurations	73
Creating a New Configuration	74
Setting Build Settings for a Configuration	75
Debugging	76
Just-in-Time Debugging	77
Standard Debugging -- Breakpoints	78
Standard Debugging -- Watch Variables	79
Debugger Options	80
Stepping with the Debugger	81
Demo: Stepping with the Debugger	82
The Call Stack	83
Summary	84
Chapter 4 Data Types in C#	85
Strong Typing	87
Demo: Typing in C#, VB and C++	88
C# Types	89
Integer Types	90
Integer Type Range	91
Integer Literals	92
Floating Point Types	93
Floating Point Literals	94
IEEE Standard for Floating Point	95
Decimal Type	96
Decimal Literals	97
Character Type	98
Character Literals	99
Escape Characters	100

Boolean Type	101
Implicit Conversions	102
Explicit Conversions	103
Conversions Example	104
Lab 4	105
Summary	106
Chapter 5 Operators and Expressions.....	111
Operator Cardinality	113
Arithmetic Operators	114
Multiplication.....	115
Division.....	116
Additive Operators.....	117
Increment and Decrement	118
Example: A Small Calculator	119
Relational Operators	120
Conditional Logical Operators.....	121
Short-Circuit Evaluation	122
Ternary Conditional Operator	123
Bitwise Operators.....	124
Bitwise Logical Operators.....	125
Bitwise Shift Operators.....	126
Assignment Operators.....	127
Expressions	128
Precedence	129
Associativity.....	130
Checking	131
Lab 5	132
Summary	133
Chapter 6 Control Structures	137
If Test	139
Blocks	140
Loops.....	141
While Loop	142
Do/While Loops.....	143
For Loops	144
Foreach Loop	145
Control Flow – Break and Continue	146
goto	147
Switch	148
Switch in C# and C/C++	149
Lab 6A.....	150
Lab 6B.....	151
Summary	152

Chapter 7 Object-Oriented Programming	157
Objects	159
Objects in the Real World	160
Object Models	161
Reusable Software Components	162
Objects in Software	163
State and Behavior	164
Abstraction	165
Encapsulation	166
Classes	167
Inheritance Concept	168
Inheritance Example	169
Relationships Among Classes	170
Polymorphism	171
Object Oriented Analysis and Design	173
Use Cases	174
CRC Cards and UML	175
Summary	176
Chapter 8 Classes	177
Classes As Structured Data	179
Classes and Objects	180
References	181
Instantiating and Using an Object	182
Assigning Object References	183
Garbage Collection	184
Sample Program	185
Methods	186
Method Syntax Example	187
Public and Private	188
Abstraction	190
Encapsulation	191
Initialization	192
Initialization with Constructors	193
Default Constructor	195
this	196
TestAccount Sample Program	197
Static Fields And Methods	199
Static Methods	200
Sample Program	201
Static Constructor	202
Constant And Readonly Fields	203
Lab 8	204
Summary	205
Chapter 9 The C# Type System	209

Overview Of Types In C#	211
Value Types	212
Simple Types.....	213
Types in System Namespace.....	214
Structures	215
Uninitialized Variables	216
Test Program	217
Copying a Structure	218
Classes and Structs.....	219
Enumeration Types	220
Enumeration Types Examples	221
Reference Types.....	222
Class Types	223
object.....	224
string	225
Arrays.....	226
Interfaces.....	227
Delegates.....	228
Default Values	229
Boxing And Unboxing.....	231
Lab 9	232
Summary	233
Chapter 10 Methods, Properties, and Operators.....	239
Static and Instance Methods	241
Method Parameters	242
No “Freestanding” Functions in C#.....	243
Classes with All Static Methods	244
Parameter Passing.....	245
Parameter Terminology.....	246
Value Parameters	247
Reference Parameters.....	248
Output Parameters.....	251
Structure Parameters	252
Class Parameters	253
Method Overloading	254
>> Method Overloading (Cont’d).....	255
Lab 10A.....	256
Modifiers as Part of the Signature.....	257
Variable Length Parameter Lists.....	258
Properties	259
Properties Example	260
Lab 10B.....	263
Operator Overloading	264
Sample Program.....	267

Operator Overloading in the Class Library	268
Summary	269
Chapter 11 Characters and Strings.....	277
Characters	279
Sample Program	280
Character Codes	281
ASCII and Unicode	282
Escape Sequences	283
Strings	284
String Class	285
Compiler Support.....	286
String Literals and Initialization	287
Concatenation	288
Index	289
Relational Operators	290
String Equality	291
String Comparisons.....	292
String Comparison	293
String Input	295
String Methods and Properties.....	296
StringBuilder Class	298
StringBuilder Equality	300
Programming With Strings	301
Command Line Arguments.....	302
Command Line Arguments in the IDE	303
Command Loops.....	304
Splitting a String	305
Lab 11	306
Summary.....	307
Chapter 12 Arrays and Indexers	311
Arrays.....	313
One Dimensional Arrays.....	314
System.Array.....	315
Sample Program.....	316
Interfaces for System.Array	317
Random Number Generation	318
Constructors for Random.....	319
Next Methods.....	320
agged Arrays	321
Rectangular Arrays.....	322
Arrays As Collections	323
Bank Case Study: Step 1	325
Account Class	327
Bank Class	330

TestBank Class.....	333
Atm Class.....	335
Running the Case Study.....	337
Indexers.....	339
ColorIndex Example Program	341
Lab 12	342
Summary	343
Chapter 13 Inheritance.....	349
Inheritance Fundamentals	351
Inheritance in C#.....	352
Single Inheritance	353
Root Class – <i>Object</i>	354
Access Control.....	355
Public Class Accessibility.....	356
Internal Class Accessibility.....	357
Member Accessibility	358
Member Accessibility Qualifiers	359
Member Accessibility Example.....	360
Method Hiding.....	361
Method Hiding and Overriding.....	362
Example: Method Hiding.....	363
Initialization	364
Initialization Fundamentals.....	365
Initialization Fundamentals Example	366
Default Constructor.....	367
Overloaded Constructors	368
Example: Overloaded Constructors	369
Invoking Base Class Constructors	370
Bank Case Study: Step 2.....	371
Bank Case Study Analysis	372
Account.....	373
CheckingAccount.....	374
SavingsAccount	376
TestAccount	378
Running the Case Study.....	379
Lab 13	380
Summary	381
Chapter 14 Virtual Methods and Polymorphism.....	385
Introduction to Polymorphism	387
Abstract and Sealed Classes.....	388
Virtual Methods And Dynamic Binding.....	389
Type Conversions in Inheritance	390
Converting Down the Hierarchy	391
Converting Up the Hierarchy	392

Virtual Methods	393
Virtual Method Example.....	394
Virtual Method Cost	395
Method Overriding.....	396
The Fragile Base Class Problem	397
<i>override</i> Keyword.....	398
Polymorphism	399
Polymorphism Using “Type Tags”	400
Polymorphism Using Virtual	401
Polymorphism Example.....	402
Abstract Classes	406
Keyword: abstract	407
Sealed Classes	408
Heterogeneous Collections	409
Heterogeneous Collections Example	410
Bank Case Study: Step 3	411
Case Study Classes.....	412
Run the Case Study.....	414
Account.....	415
CheckingAccount, SavingsAccount.....	416
Bank and Atm	417
TestBank	418
Lab 14	419
Summary	420
Chapter 15 Formatting and Conversion	427
Introduction to Formatting	429
ToString	430
ToString in Your Own Class.....	431
Using Placeholders.....	433
Format Strings.....	434
Simple Placeholders.....	435
Controlling Width	436
Format String	437
Currency.....	438
Currency Format Example	439
String.Format	440
PadLeft and PadRight	441
Bank Case Study: Step 4	443
Type Conversions	444
Conversion of Built-In Types.....	445
Conversion of User-Defined Types	446
User Defined Conversions: Example.....	448
Lab 15	450
Summary	451

Chapter 16 Exceptions.....	455
Introduction to Exceptions	457
Exception Fundamentals	458
.NET Exception Handling	459
Exception Flow of Control.....	460
Context and Stack Unwinding	461
Exception Example	462
System.Exception.....	465
User-Defined Exception Classes.....	466
User Exception Example.....	467
Structured Exception Handling.....	470
Finally Block.....	471
Inner Exceptions	473
Checked Integer Arithmetic	474
Example Program.....	475
Lab 16	476
Summary	477
Chapter 17 Interfaces	481
Introduction.....	483
Interfaces in C#	485
Interface Inheritance.....	486
Programming With Interfaces	487
Implementing Interfaces.....	488
Using an Interface	490
Demo: SmallInterface	491
Dynamic Use Of Interfaces	492
Demo: TryInterfaces	493
is Operator.....	494
as Operator	495
Bank Case Study: Step 6.....	496
Common Interfaces in Case Study--IAccount	497
Apparent Redundancy	498
IStatement	499
IStatement Methods	500
IChecking.....	501
ISavings.....	502
The Implementation	503
SavingsAccount	504
The Client.....	505
Resolving Ambiguity	507
Access Modifier	508
Summary	509
Chapter 18 Interfaces and the .NET Framework	511
Overview	513

Collections	514
ArrayList Example	515
Count and Capacity	516
foreach Loop	517
Array Notation	518
Adding to the List	519
Remove Method	520
RemoveAt Method	521
Collection Interfaces	522
IEnumerable and IEnumerator	523
IEnumerable and IEnumerator Demo: <i>AccountList</i>	524
ICollection	525
IList	526
A Collection of User-Defined Objects	527
A Correction to AccountList (Step 1)	528
Bank Case Study: Step 7	529
Copy Semantics And ICloneable	530
Copy Semantics in C#	531
Shallow Copy and Deep Copy	532
Example Program	533
Reference Copy	534
Memberwise Clone	535
Using ICloneable	536
Comparing Objects	537
Sorting an Array	538
Anatomy of Array.Sort	539
Using the is Operator	540
The Use of Dynamic Type Checking	541
Implementing IComparable	542
Complete Solution	543
Understanding Frameworks	544
Lab 18	545
Summary	546
Chapter 19 Delegates and Events	553
Overview of Delegates and Events	555
Callbacks and Delegates	556
Usage of Delegates	557
Declaring a Delegate	558
Defining a Method	559
Creating a Delegate Object	560
Calling a Delegate	561
Combining Delegate Objects	562
Complete Example	563
Stock Market Simulation	564

Running the Simulation	566
Delegate Code	567
Passing the Delegates to the Engine	568
Using the Delegates	569
Events.....	570
Events in C# and .NET	572
Server-Side Event Code.....	573
Delegate Objects	574
Client-Side Event Code.....	575
Chat Room Example	576
Client Code	577
Server Code.....	578
Summary	579
Chapter 20 Introduction to Windows Forms	581
Creating a Windows Forms App.....	583
Windows Forms Event Handling.....	587
Add Events for a Control	588
Events Documentation	589
Lab 20	590
Summary	591
Appendix A Learning Resources.....	621

Chapter 4

Data Types in C#

Data Types in C#

Objectives

After completing this unit you will be able to:

- **Explain what "Strongly Typed" means, and why strongly-typed languages contribute to program reliability.**
- **Explain when implicit conversion is used in C#, and when casting must be used.**
- **Name the types available in C#, and state the size of each type.**
- **Use C# types in simple programs.**

Strong Typing

- **C# is a *strongly typed* language.**
 - Variables and constants of different types may not be mixed, except according to strict rules.
 - The C# compiler will refuse to compile type conversions that are not 'safe'.
- **By contrast, C++ is a weakly typed language, and Visual Basic is untyped.**
- **Strong typing prevents certain types of errors**
 - Unintended mixing of types
 - Loss of precision

Demo: Typing in C#, VB and C++

- **C# is a strongly typed language.**
 - See **Seven\Charp**
- **C++ is a weakly typed language.**
 - See **Seven\Cpp**
- **VB6 can be programmed in an untyped manner.**
 - See **Seven\Vb**

C# Types

- **In C#, the size of each type is specified.**
 - In C and C++, the sizes of the types are not completely specified; only relative sizes are given.
 - C# is similar to Java in size specification.
- **C# has five categories of types.**
 - Integer Types
 - Floating-point Types
 - Decimal Type
 - Character Type
 - Boolean Type

Integer Types

- **C# has a large variety of integer data types.**
 - Allows great flexibility to choose appropriate size and signed/unsigned for you application.

C# Keyword	Size	Signed/ Unsigned	Type in System Namespace
sbyte	8 bits	signed	SByte
byte	8 bits	unsigned	Byte
short	16 bits	signed	Int16
ushort	16 bits	unsigned	UInt16
int	32 bits	signed	Int32
uint	32 bits	unsigned	UInt32
long	64 bits	signed	Int64
ulong	64 bits	unsigned	UInt64

Integer Type Range

- Use the *MinValue* and *MaxValue* members of the types in the *System* namespace to find the range of each integer type.

– Example:

```
Console.WriteLine("min int= " + UInt32.MinValue);  
Console.WriteLine("max int= " + UInt32.MaxValue);
```

– Output:

```
min int = -2147483648  
max int = 2147483647
```

– See the program **IntegerRange**

Integer Literals

- **A literal is a source code representation of a value.**

```
squaresChess = 64;
```

- In this example, '64' is an integer literal.

- **Integer literals are always stored as either 32-bit or 64-bit values.**

- In the above example, the value '64' is stored as a 32-bit number.

- **You may specify the type that is used with a suffix.**

- L (or l) for long
- U (or u) for unsigned
- Suffixes may be combined: LU, UL, ul, lu (any combination of case and order)

- **Hexadecimal representation uses the prefix 0x (or 0X).**

Floating Point Types

- Used to express very large and very small quantities.
- There are two floating point types in C#, corresponding to single and double precision.

C# Keyword	Size	Type in System Namespace
float	32 bits	Single
double	64 bits	Double

- There is actually a third floating point type, called *decimal*, which is covered later in this module.
- Use the *MinValue* and *MaxValue* members of the types in the *System* namespace to find the range of each floating point type.
 - See Lab 4

Floating Point Literals

- **Floating point literals may use either decimal or exponential notation.**
 - Decimal notation example: 3.141529
 - Exponential notation example: 2e-9 (0.000000002)
 - Mixed notation: 3.01e3 (3,010)
- **The default type of a floating point literal is double.**
 - To specify float, use a suffix of **F** or **f**.
 - A suffix of **D** or **d** can be used to signify double (even though that is redundant, you might want to emphasize it for some reason).

IEEE Standard for Floating Point

- **C# uses the IEEE 754 floating point standard (see <http://grouper.ieee.org/groups/754/>).**
 - IEEE 754-1985 governs binary floating-point arithmetic. It specifies number formats, basic operations, conversions, and exceptional conditions. The related standard IEEE 854-1987 generalizes 754 to cover decimal arithmetic as well as binary.
- **IEEE 754 defines two special values, NaN, and Infinity**
 - Infinity results from an attempt to divide a non-zero number by a floating point zero.
 - NaN results from an attempt to divide a floating point zero by any number, including floating point zero.
- **Example program**
 - See **SpecialFloat**

Decimal Type

- **Floating point types (even double) are frequently not sufficiently precise for financial calculations.**
 - Even a small round-off error may be unacceptable.
- **To address this problem, a new type was introduced which can precisely represent decimal numbers with up to 28 digits.**

C# Keyword	Size	Type in System Namespace
decimal	96 bits	Decimal

- **As with integer and floating point types, the range may be found using the *MinValue* and *MaxValue* member functions of *Decimal*.**
- **This type is new for C#.**
 - C++ and Java require library support to get the equivalent of decimal.

Decimal Literals

- **The decimal literal suffix is M**
 - For "Money".
 - The M suffix may be combined with exponential and decimal notation.

```
decimal billGatesSalary = 4.0e6M;
```

- **Example program:**
 - See **DecimalLiterals**

Character Type

- **C# uses a 16-bit Unicode character set.**

C# Keyword	Size	Type in System Namespace
char	16 bits	Char

Character Literals

- **Character in single quotes**
 - `'7'` // Unicode character '7'
- **Hex encoding (\x)**
 - `\x0055` // '7' in hex
- **Unicode prefix (\u)**
 - `\u0055` // '7' using Unicode prefix
- **To represent characters with a special meaning, use a backslash (\) to 'escape'.**
 - `'\"` // the single quote character
 - `'\\'` // the backslash character
- **Example program:**
 - See `CharacterLiterals`

Escape Characters

- **There are a number of non-printing or special characters used by C# which have been given special escape sequences.**

Escape Character	Name	Value
\'	Single quote	0x0027
\"	Double quote	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009
\v	Vertical tab	0x000B

Boolean Type

- The **bool** type represents the logical values 'true' and 'false'.
 - **true** and **false** are the two Boolean literals.

C# Keyword	Size	Type in System Namespace
bool	8 bits	Boolean

Implicit Conversions

- Since C# is *strongly typed*, if we need to use a value of one data type where another type is expected, we must use a *conversion*.
- An *implicit* conversion is done silently by the compiler where needed.
 - Implicit conversions are only done for *safe* conversions, where the target type has a larger dynamic range (wider) than the type to be converted.
- **Examples:**
 - float to double
 - byte to int

Explicit Conversions

- An *explicit* conversion is specified by the programmer, using a *cast*.
 - An explicit conversion is generally required if the target type has a smaller dynamic range (narrower).

– Example:

```
float pi = 3.141529; // compiler error;  
                        // default is double  
float pi = (double) 3.141529; // ok
```

- An explicit conversion is needed if the dynamic range of the target doesn't include all of the values possible for the type to be converted.

– Example:

```
char seven = '7';  
short number;  
number = seven; // compiler error  
// character range 0 to 65535  
// short range -32768 to 32767
```

- The *bool* type may not be cast to or from any other type.
 - However, there are conversion functions available in a special class (called **Convert**) for this purpose.

Conversions Example

- **Example program illustrates a number of issues in implicit and explicit conversion.**

```
// Conversions.cs

using System;

public class Conversions
{
    public static int Main(string[] args)
    {
        // float pi = 3.14;          // compiler error
        float pi = (float) 3.14;
        Console.WriteLine("pi = " + pi);
        // short seven = '7';        // compiler error
        short seven = (short) '7';    // cast
        ushort useven = '7';          // ok
        Console.WriteLine("seven = " + seven);
        Console.WriteLine("useven = " + useven);
        //int itrue = (int) true;     // cast fails
        int itrue = Convert.ToInt32(true);
        int ifalse = Convert.ToInt32(false);
        Console.WriteLine("itrue = " + itrue);
        Console.WriteLine("ifalse = " + ifalse);
        return 0;
    }
}
```

Lab 4

Type Sizes

In this lab, you will investigate the sizes of the different types. You will write a C# program to find and display the largest or smallest values of various data types. You will also modify a program to properly calculate a large integer value, corresponding to a terabyte.

Detailed instructions are contained in the Lab 4 write-up at the end of the chapter.

Suggested time: 20 minutes

Summary

- **C# is strongly typed**
- **Types in C# have specified sizes**
- **Some conversions are safe, and can be implicitly performed by the compiler.**
- **Some conversions are potentially unsafe, but may be explicitly specified by the programmer, using casts**
- **Boolean conversions to or from other types are only permitted using member conversion functions**

Lab 4

Type Sizes

Introduction

In this lab, you will investigate the sizes of the different types. You will write a C# program to find and display the largest or smallest values of various data types. You will also modify a program to properly calculate a large integer value, corresponding to a terabyte.

Suggested Time: 20 minutes

Root Directory: OIC\CSharp

Directories:	Labs\Lab4\LargeSmall	(Exercise 1 work area)
	Chap04\LargeSmall	(Exercise 1 answer)
	Labs\Lab4\Bytes	(Exercise 2 work area)
	Chap04\Bytes\Step1	(Exercise 2 backup of starter files)
	Chap04\Bytes\Step2	(Exercise 2 answer)

Exercise 1. Large and Small Numbers

1. Which of the following data types can represent the largest positive number?

float, double, decimal
2. Which of the following data types can represent a number with the greatest precision?
Which would have the least precision?

float, double, decimal
3. Write a C# program that will find the following:
 - a. The largest float
 - b. The smallest decimal
 - c. The smallest double

Your program should print lines that look like:

The largest float is xxxxxxxxx

Use Visual Studio to create an empty C# project **LargeSmall** in the **Lab4** folder. This will create the folder **LargeSmall**. Add a new file **LargeSmall.cs** to your project, where you will place your program code.

Exercise 2. Calculating Bytes

Build and run the starter program, which prints out the number of bytes in a kilobyte, megabyte and gigabyte. Note that this program does no input. Try making a simple extension to add the declaration of an **int** variable **tera**, which you will use to hold the number of bytes in a terabyte (approximately a trillion bytes, or 1024 times a gigabyte). The program fails. Why? Fix the problem!

The proper output is:

```
1 kilobyte = 1024 bytes
1 megabyte = 1048576 bytes
1 gigabyte = 1073741824 bytes
1 terabyte = 1099511627776 bytes
```

Lab 4 Answers

Exercise 1

1. The largest positive number can be represented by **double**.
2. The highest precision can be achieved by using **decimal**, the lowest by **float**.
3. See the program **LargeSmall**.

Exercise 2

The problem is that **int** cannot hold a large enough integer. The fix is to simply use the **long** data type, as shown in **Bytes\Step2**.

```
// Bytes.cs - Step 2
//
// Calculate and print out number of bytes in a kilobyte,
// megabyte, and gigabyte
// Step 2 -- terabyte, using long

using System;

class Bytes
{
    public static void Main(String[] args)
    {
        int kilo = 1024;
        int mega = kilo * kilo;
        long giga = kilo * mega;
        long tera = kilo * giga;
        Console.WriteLine("1 kilobyte = " + kilo + " bytes");
        Console.WriteLine("1 megabyte = " + mega + " bytes");
        Console.WriteLine("1 gigabyte = " + giga + " bytes");
        Console.WriteLine("1 terabyte = " + tera + " bytes");
    }
}
```


Chapter 16

Exceptions

Exceptions

Objectives

After completing this unit you will be able to:

- **Use the C# exception mechanism.**
- **Create and use your own exception classes.**

Introduction to Exceptions

- **An inevitable part of programming is dealing with error conditions of various sorts.**
- **This chapter introduces the exception handling mechanism of C#, beginning with a discussion of the fundamentals of error processing and various alternatives that are available.**
- **The .NET class library provides an *Exception* class, which you can use to pass information about an exception that occurred.**
- **To further specify your exception and to pass additional information, you can derive your own class from *Exception*.**
- **When handling an exception you may want to throw a new exception.**
- **In such a case you can use the “inner exception” feature of the *Exception* class to pass the original exception on with your new exception.**
- **We will illustrate these features with a simplified version of our case study example, and then provide an update to the case study itself, incorporating basic exception handling.**
- **We also provide an example of handling arithmetic exceptions.**

Exception Fundamentals

- **The traditional way to deal with errors when programming is to have the functions you call return a status code.**
 - The status code may have a particular value for a good return and other values to denote various error conditions.
 - The calling function checks this status code, and if an error was encountered, it performs appropriate error handling.
 - This function in return may pass an error code to its calling function, and so on up the call stack.
- **Although straightforward, this mechanism has a number of drawbacks, principally lack of robustness.**
 - The called function may have impeccable error checking code and return appropriate error information, but all this information is wasted if the calling function does not make use of it.
 - The program may continue operation as if nothing were amiss, and sometime later, crash for some mysterious reason.
 - Another disadvantage is that every function in the call stack must participate in the process, or the chain of error information will be broken.
 - In languages such as C# that have constructors and overloaded operators, there is not even a return value for some operations.

.NET Exception Handling

- **C# provides an *exception* mechanism that can be used for reporting and handling errors.**
 - An error is reported by “throwing” an exception.
 - The error is handled by “catching” the exception.
 - This mechanism is similar in concept to exceptions in C++ and Java.
 - Exceptions are implemented in .NET by the Common Language Runtime, so exceptions can be thrown in one .NET language and caught in another.
- **The exception mechanism involves the following elements:**
 - Code that might encounter an exception should be enclosed in a **try** block.
 - Exceptions are caught in a **catch** block.
 - An Exception object is passed as a parameter to catch. The data type is **System.Exception** or a derived type.
 - You may have multiple **catch** blocks. A match is made based on the data type of the **Exception** object.
 - An optional **finally** clause contains code that will be executed whether or not an exception is encountered.
 - In the called method, an exception is raised through a **throw** statement.

Exception Flow of Control

- **The general structure of code which might encounter an exception is shown below:**

```
try
{
    // code that might throw an exception
}
catch (ExceptionClass1 e)
{
    // code to handle this type of exception
}
catch (ExceptionClass2 e)
{
    // code to handle this other type of exception
}
// possibly more catch handlers
// optional finally clause (discussed later)
// statements after try ... catch
```

- **Each catch handler has a parameter specifying the data type of exception that it can handle.**
- **The exception data type can be *System.Exception* or a class ultimately derived from it.**
 - If an exception is thrown, the *first* catch handler that matches the exception data type is executed, and then control passes to the statement just after the catch block(s).
 - If no handler is found, the exception is thrown to the next higher “context” (e.g., the function that called the current one). If no exception is thrown inside the **try** block, all the catch handlers are skipped.

Context and Stack Unwinding

- **As the flow of control of a program passes into nested blocks, local variables are pushed onto the stack and a new “context” is entered.**
 - Likewise, a new context is entered on a method call, which also pushes a return address onto the stack.
 - If an exception is not handled in the current context, the exception is passed to successively higher contexts until it is finally handled (or else is “uncaught” and is handled by a default system handler).
- **When the higher context is entered, C# adjusts the stack properly, a process known as *stack unwinding*.**
 - In C# exception handling, stack unwinding involves both setting the program counter and cleaning up variables (popping stack variables and marking heap variables as free, so that the garbage collector can deallocate them).

Exception Example

- Now let's look at some code that illustrates the principles we have discussed so far.
- We will use a simplified version of our *Account* class, which has only methods *Deposit* and *Withdraw*, and the property *Balance*.
 - Both methods will throw an exception if the amount passed as a parameter is negative.
 - In addition, the **Withdraw** method will throw an exception if the new balance would be negative—overdrafts are not allowed.
- Our example program is in the directory *AccountExceptionDemo\Step1*.
- In the test program, we place the entire body of the command processing loop inside a *try* block.
 - The catch handler prints an error message that is passed within the exception object.
 - Then after either normal processing or displaying an error message, a new command is read in.
 - This simple scheme provides reasonable error processing, as a bad command will not be acted upon, and the user will have an opportunity to enter a new command.

Exception Example (Cont'd)

```
// Account.cs

using System;

public class Account
{
    protected decimal balance;
    public Account(decimal balance)
    {
        this.balance = balance;
    }
    public void Deposit(decimal amount)
    {
        if (amount < 0.00m)
            throw new Exception(
"The transaction amount cannot be negative.");
        balance += amount;
    }
    public void Withdraw(decimal amount)
    {
        if (amount < 0.00m)
            throw new Exception(
"The transaction amount cannot be negative.");
        decimal newbal = balance - amount;
        if (newbal < 0.00m)
            throw new Exception(
"The balance cannot be negative.");
        balance = newbal;
    }
    public decimal Balance
    {
        get
        {
            return balance;
        }
    }
}
```

Exception Example (Cont'd)

```
// AccountExceptionDemo.cs

while (! cmd.Equals("quit"))
{
    try
    {
        if (cmd.Equals("deposit"))
        {
            decimal amount = iw.getDecimal(
                "amount: ");
            acc.Deposit(amount);
            ShowBalance(acc);
        }
        else if (cmd.Equals("withdraw"))
        {
            decimal amount = iw.getDecimal(
                "amount: ");
            acc.Withdraw(amount);
            ShowBalance(acc);
        }
        else if (cmd.Equals("show"))
            ShowBalance(acc);
        else
            help();
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
    cmd = iw.getString("> ");
}
```

System.Exception

- The *System.Exception* class provides a number of useful methods and properties for obtaining information about an exception.
 - **Message** returns a text string providing information about the exception.
 - This message is set when the exception object is constructed.
 - If no message is specified, a generic message will be provided, indicating the type of the exception.
 - The **Message** property is read-only. (Hence, if you want to specify your own message, you must construct a new exception object, as done in the example above.)
 - **StackTrace** returns a text string providing a stack trace at the place where the exception arose.
 - **InnerException** holds a reference to another exception.
 - When you throw a new exception, it is desirable not to lose the information about the original exception.
 - The original exception can be passed as a parameter when constructing the new exception.
 - The original exception object is then available through the **InnerException** property of the new exception. (We will provide an example of using inner exceptions later in this chapter.)

User-Defined Exception Classes

- **You can do basic exception handling using only the base *Exception* class, as illustrated previously.**
 - In order to obtain more fine-grained control over exceptions, it is frequently useful to define your own exception class, derived from **Exception**.
 - You can then have a more specific catch handler that looks specifically for your exception type.
 - You can also define other members in your derived exception class, so that you can pass additional information to the catch handler.
- **We will illustrate by enhancing the *Withdraw* method of our *Account* class (*AccountExceptionDemo\Step1*).**
 - We want to distinguish between the two types of exceptions we throw. The one type is essentially bad input data (a negative value).
 - We will continue to handle this exception in the same manner as before (which is the same as bad input data that gives rise to a format exception, thrown by .NET library code).
 - We will define a new exception class **BalanceException** to cover the case when the balance would become negative.
 - In this case we want to allow the user an opportunity to correct the situation (in this case simply by making a deposit to cover the shortage).

User Exception Example

- Our example program is *AccountExceptionDemo\Step2*.
- Note that we define a property *Shortage* that can be used to store the information about how short the balance is.
 - The constructor of our exception class takes two parameters.
 - The first parameter is an error message string, and the second parameter is the amount of the shortage.
 - We pass the message string to the constructor of the base class.
 - We must also modify the code of the **Account** class to throw our new type of exception when an illegal negative balance would be created.
- Finally we modify the code in our test program that processes the “withdraw” command.
 - We place the call to **Withdraw** inside another **try** block, and we provide a catch handler for a **BalanceException**. In this catch handler we allow the user an opportunity to make a supplemental deposit.

User Exception Example (Cont'd)

```
// BalanceException.cs

using System;

public class BalanceException : Exception
{
    private decimal shortage;
    public BalanceException(string message,
        decimal shortage) : base(message)
    {
        this.shortage = shortage;
    }
    public decimal Shortage
    {
        get
        {
            return shortage;
        }
    }
}

// Account.cs

...

public void Withdraw(decimal amount)
{
    if (amount < 0.00m)
        throw new Exception(
"The transaction amount cannot be negative.");
    decimal newbal = balance - amount;
    if (newbal < 0.00m)
        throw new BalanceException(
"The balance cannot be negative.", -newbal);
    balance = newbal;
}
```

User Exception Example (Cont'd)

```
// AccountExceptionDemo.cs

...

else if (cmd.Equals("withdraw"))
{
    decimal amount = iw.getDecimal("amount: ");
    try
    {
        acc.Withdraw(amount);
    }
    catch (BalanceException e)
    {
        Console.WriteLine("You are short {0:C}",
            e.Shortage);
        Console.WriteLine("Please make a deposit");
        decimal supplemental = iw.getDecimal(
            "amount: ");
        acc.Deposit(supplemental);
        acc.Withdraw(amount);    // try again
    }
}
```


Structured Exception Handling

- **One of the principles of structured programming is that a block of code should have a single entry point and a single exit point.**
 - The **goto** statement is bad, because it facilitates breaking this principle.
 - But there are other ways to violate the principle of a single exit point, such as multiple **return** statements from a method.
 - Multiple return statements may not be too bad, because these may be encountered during normal, anticipated flow of control.
- **Exceptions can cause a particular difficulty, since they interrupt the normal flow of control.**
- **In a common scenario you can have at least three ways of exiting a method:**
 - No exception is encountered, and any catch handlers are skipped.
 - An exception is caught, and control passes to the code after the catch handlers.
 - An exception is caught, and the catch handler itself throws another exception. Then code after the catch handler will be bypassed.
- **The first two cases aren't a problem, as in both cases control passes to the code after the catch handlers, but the third case is a source of difficulty.**

Finally Block

- **The structured exception handling mechanism in C# resolves this problem with a *finally* block.**
 - The **finally** block is optional, but if present must appear immediately after the **catch** handlers.
 - It is guaranteed that the code in the **finally** block will always execute before the method is exited, in all three cases described.
- **We illustrate use of finally in the *Withdraw* command of our *Account* example.**
- **See the directory *AccountExceptionDemo\Step3*.**
 - There are several ways to exit this block of code, and the user might become confused about her balance upon exiting.
 - We insert a **finally** block, which will display the balance.

Finally Block (Cont'd)

```
// AccountExceptionDemo.cs

else if (cmd.Equals("withdraw"))
{
    decimal amount = iw.getDecimal("amount: ");
    try
    {
        acc.Withdraw(amount);
    }
    catch (BalanceException e)
    {
        Console.WriteLine(
            "You are short {0:C}", e.Shortage);
        Console.WriteLine("Please make a deposit");
        decimal supplemental = iw.getDecimal(
            "amount: ");
        acc.Deposit(supplemental);
        acc.Withdraw(amount);    // try again
    }
    finally
    {
        ShowBalance(acc);
    }
}
```

Inner Exceptions

- **In general, it is most convenient to handle exceptions near their source, because you have the most information available about the context in which the exception occurred.**
 - A common pattern is to create a new exception object that captures more detailed information and throw this on to the calling program.
- **When you throw a new exception, you don't want to lose the information about the original exception.**
- **The original exception can be passed as a parameter when constructing the new exception.**
 - The original exception is then available through the **InnerException** property of the new exception.
 - Notice that in the code above we pass the exception object *e* as a parameter to the constructor of the new **Exception** object that we throw.
- **In the *ArithmeticExceptionDemo* program we review checked integer arithmetic (you may wish to refer to the last section of Chapter 5), and then we present the entire example program.**

Checked Integer Arithmetic

- By default in C#, integer overflow does not raise an exception; instead the result is truncated.
- The *checked* operator will cause the integer calculation to check for overflow and throw an exception if an overflow condition arises.
- You can cause all integer arithmetic to be checked via the */checked* compiler command line switch.
- You can turn off checking by the *unchecked* operator.
- Unchecked arithmetic is faster but less safe.

Example Program

- The *ArithmeticExceptionDemo* program demonstrates a number of scenarios of arithmetic exceptions.
 - You can experiment by commenting and uncommenting different sections of code.
 - You can also try building with the **/checked** compiler option.
 - Notice how in the main program we display the inner exception, if any. (If there is no inner exception, the **InnerException** property will be **null**.)

```
public static int Main(string[] args)
{
    int prod;
    long lprod;
    try
    {
        lprod = LongMultiply(56666L, 57777L);
        Console.WriteLine("product = {0}", lprod);
        prod = Multiply(56666, 57777);
        Console.WriteLine("product = {0}", prod);
        ...
    }
    catch (Exception e)
    {
        Console.WriteLine("Exception: {0}",
            e.Message);
        if (e.InnerException != null)
            Console.WriteLine("Inner Exception: {0}",
                e.InnerException.Message);
    }
    return 0;
}
```

Lab 16

Time Exceptions

The **Time** class from Lab 15 does not do any error checking when a string is converted into a time. In this lab you will add basic error checking by throwing an exception if the string cannot be converted to a valid time. Note that the conversion is done by an operator, so exception handling is the only mechanism you can reasonably use; a status return value will not work.

Detailed instructions are contained in the Lab 16 write-up at the end of the chapter.

Suggested time: 30 minutes

Summary

- The traditional way to deal with errors in programs is to check an error return code.
- This approach has a number of defects, the most important of which is that the calling program may simply ignore error returns.
- C# provides an exception mechanism, which includes a *try* block, *catch* handlers, and a *finally* block.
- You can raise exceptions by means of a *throw* statement.
- The .NET class library provides an *Exception* class, which you can use to pass information about an exception that occurred.
- To further specify your exception and to pass additional information, you can derive your own class from *Exception*.
- When handling an exception, you may want to throw a new exception.
- In such a case you can use the “inner exception” feature of the *Exception* class to pass the original exception on with your new exception.

Lab 16

Time Exceptions

Introduction

The **Time** class from Lab 15 does not do any error checking when a string is converted into a time. In this lab you will add basic error checking by throwing an exception if the string cannot be converted to a valid time. Note that the conversion is done by an operator, so exception handling is the only mechanism you can reasonably use; a status return value will not work.

Suggested Time: 30 minutes

Root Directory: OIC\CSharp

Directories:	Labs\Lab16\TimeException	(Work area)
	Chap16\TimeException\Step1	(Backup of starter files)
	Chap16\TimeException\Step2	(Answer)

Instructions

1. Build the starter code and run it a few times with valid and invalid input data. Note how the program crashes on invalid input.
2. Modify the **Time** class to throw exceptions for the following conditions:
 - a. User inputs a string not of the form of two strings separated by a colon.
 - b. The hours are not between 0 and 23.
 - c. The minutes are not between 0 and 59.
 - d. Any other errors you discover (the answer given may not be complete).
3. Build and run with the original test program. It still crashes, but if you inspect the error message, you should see that your various exceptions were thrown.
4. Add exception handling to the test program to catch exceptions and display an error message, and allow the user to enter another time string.
5. Build and run your new test program. Test with both data you validate for, and for other invalidate data (for example, two non-numeric strings separated by a colon). In the latter case you should see an exception thrown by the system.

Lab 16 Answer

```
// Time.cs

using System;

public class Time
{
    private int hour;
    private int minute;
    public Time(int h, int m)
    {
        hour = h;
        minute = m;
    }
    public static explicit operator Time(string str)
    {
        char[] seps = {':'};
        string[] pieces = str.Split(seps);
        if (pieces.Length != 2)
            throw new Exception("Improper format for time");

        int hour = Convert.ToInt32(pieces[0]);
        if (hour < 0 || hour > 23)
            throw new Exception("Hours out of range");

        int minute = Convert.ToInt32(pieces[1]);
        if (minute < 0 || minute > 59)
            throw new Exception("Minutes out of range");

        Time t = new Time(hour, minute);
        return t;
    }
    public static implicit operator string(Time t)
    {
        return t.hour + ":" + t.minute;
    }
    public void Show()
    {
        Console.WriteLine("{0} hours {1} minutes", hour, minute);
    }
}
```

```
// TestTime.cs

using System;

public class TestTime
{
    static void Main()
    {
        Console.WriteLine("Enter time in format hh:mm");
        Console.WriteLine("Press carriage return to quit");
        InputWrapper iw = new InputWrapper();
        string str = iw.getString("time: ");
        while (str != "")
        {
            try
            {
                Time t = (Time) str;
                t.Show();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
            str = iw.getString("time: ");
        }
        Console.WriteLine("Goodbye");
    }
}
```