CENTRIQ'S

TechSmartKC

Fundamentals of C# .NET Learning Guide

Table of Contents

.NET Framework Overview

Lesson Objectives

1. Basics of .NET Framework
2. The .NET Framework Components
3. Common Language Runtime (CLR)
4. .NET Framework Class Library (FCL)

Basics of .NET Platform

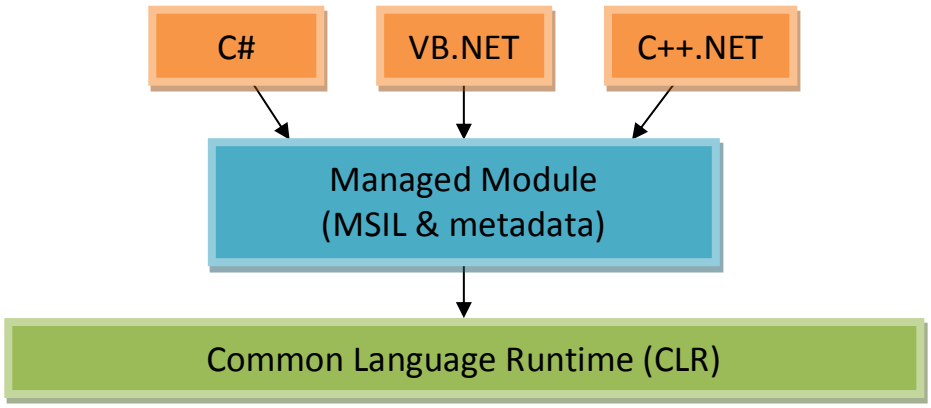
1. Independence from a specific language or platform in a new run-time environment.
2. Programming model for developing HTML pages through ASP.NET.
3. Internet servers exposing functions through Web Services.
4. Windows Forms for rich and rapid GUI development.
5. Database Access through ADO.NET.

Programmers can contribute to the same software projects using C#, Visual C++.NET, Visual Basic .NET, or other programming languages.

Web Services are applications to be used over the internet. Part of .NET includes Active Server Pages (ASP).NET for creating web applications.

Currently, .NET only executes on Windows. There is an open-source project named Mono (www.go-mono.com), which is attempting to provide the .NET Framework on Linux.

Cross Language Support



.NET provides cross language support. When you compile a .NET program, a managed module is created. A managed module is a standard Windows portable executable (PE) file containing MSIL and metadata.

MSIL is machine independent code that executes on the CLR.

The CLR uses a Just-In-Time compile to increase speed execution. The first time a method is executed it is translated into machine code and stored away for future invocations. On subsequent invocations, the CLR will use the translated machine code instead of interpreting the MSIL again.

.NET Framework

1. Framework Class Library (FCL)
2. Common Type System
3. Common Language Specification

The .NET Framework provides the necessary compile-time and run-time foundation to build .NET based applications. Currently, the .NET Framework only runs on Microsoft Windows Operating Systems, however it is expected to be extended to run on other platforms in the future. The .NET Framework provides a common type system called the unified type system. The unified type system is used by a .NET compatible language.

The Framework Class Library (FCL) is a library of classes, interfaces, and value types that are included in the Microsoft .NET Framework Standard Development Kit (SDK). This library provides access to system functionality, and is designed to be the foundation on which .NET is compatible for other platforms.

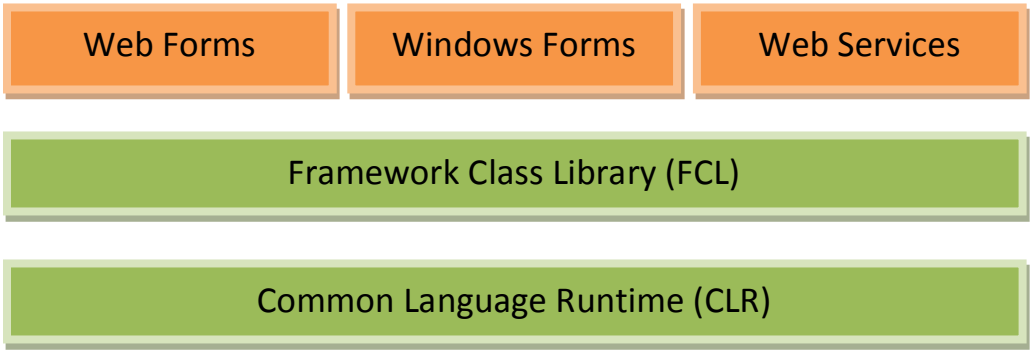
The Common Language Specification (CLS) is a set of rules that allows code that is written in different .NET languages to interact with each other.

.NET Platform

1. Microsoft Intermediate Language
2. Common Language Runtime

Microsoft Intermediate Language (MSIL or IL) is the first step in execution of source code.

The Common Language Runtime (CLR) has many duties, the most prominent being the transformation of MSIL into native (machine specific) code. This is done with a tool called the Just-In-Time compiler or JIT. CLR also manages memory, security, and other features relieving the programmer from these responsibilities.



.NET applications will generally be of one of the above: Web Forms, Windows Forms or Web Services.

1. Web Forms: provide access through web browsers.
2. Windows Forms: provide access through traditional GUI interfaces.
3. Web Services: supplies clients with data in XML format.

The Framework Class Library (FCL) includes classes used to access databases (ADO.NET), perform I/O, string manipulation, threading, networking, collections, etc. Currently, there are more than 3,400 classes in the FCL.

The Common Language Runtime (CLR) provides an environment in which .NET programs are executed.

How your C# Program Actually Runs

You start your journey towards completing and running a C# program by typing C# code. This is called your source code, and it is a similar process for anyone creating a .NET application – whether they use C# as the language of choice, or VB.NET, or C++.NET or any other .NET language. However, this language by itself cannot be used directly to run the program. When you're done typing, have saved your changes (CTRL+S) and are ready to run the program, there are still more steps to make that happen.

To move our source code forward, we need to perform the first translation, called "**building**" or "**compiling**". This process translates it to (MS)IL code and metadata, called an "**assembly**" or "**managed module**". The **IL (Intermediate Language)** is what all .NET languages must be translated to in order to eventually be run in the .NET framework. It also explains how we can build programs with different .NET languages interchangeably, since they will all be compiled into this other language anyway.

The assembly that is created represents our program, and it will be either an EXE or a DLL file. An **EXE file is a self-executing program**, meaning it is intended to be launched directly, without the pre-requisites of any other files. Keep in mind that you *can* tie an EXE to utilize other files but an EXE doesn't inherently *have to* rely on other files to run. A **DLL (Dynamic Link Library) file is a helper program** that is not capable of running independently – its job is to come when called by an EXE or another DLL. (If called by another DLL, that DLL would need to be chained directly or indirectly back to an EXE file so that some self-executing program could be independently launched to start the process.)

Why would we create a helper program when we could instead just create all our applications as self-executing? The answer: for **code re-use** and **maintainability**. If Program A and Program B both need to execute a certain process, it is often more efficient to write that process once in Program C and just let the other two programs call on Program C when needed. And if Program C's process will only ever be used in the context of other programs, it's an excellent candidate to be a DLL (helper program) instead of an EXE.

And if you further imagine that a dozen programs might make use of the functionality contained by Program C, or they might use it on multiple occasions, then this code re-use becomes exponentially more useful as a simple call to the DLL. The alternative is duplicating that code verbatim every time it is needed by any given program. Not only is this redundant and inefficient – worse, it is establishing a situation where changing or adding to the process would require careful and consistent changes to *all the places* where the redundant code had been written.

Code re-use and maintainability are the holy grails of programming. To a reasonable degree, the more we can re-use common code and maintain it well from one location, the better. It is not only more efficient, it also makes it easier to avoid buggy code (which frequently occurs from failing to consistently update all instances of the code that does essentially the same thing) and to reduce **code bloat** (the excessive addition of code that makes it harder to read or adds to the file size or processing time).

For example, imagine you are writing a Product Generator program that allows employees to input the data associated with a new product our company sells, including writing the product descriptions. A feature of the program is to allow users to spell-check that description to avoid typos. However, you also know you will be creating an Employee Evaluator program that allows managers to write up yearly reviews of employee progress and contributions. Both programs could benefit from the spell-check feature. Why write it twice into each EXE (and maintain it in both places) when you could write one DLL program called SpellChecker that could do the work when called by either Product Generator or Employee Evaluator? Besides, it's only a matter of time till you find yourself creating another program that could benefit from SpellChecker – but now it's already written and callable as a DLL.

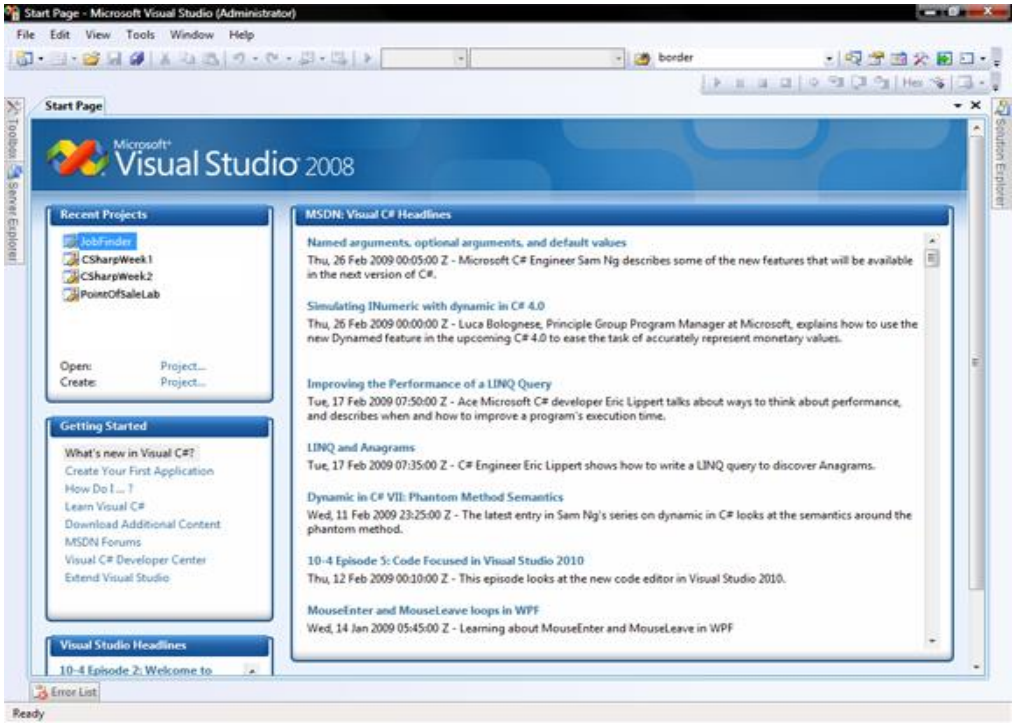
Visual Studio

Visual Studio Overview

1. Cross-Language Development Environment
2. Used to Create, Document, Run and Debug Programs

Visual Studio .NET provides the framework, compiler, and JIT to run .NET applications. Although other IDE's exist or will be produced, Visual Studio .NET will most likely remain the primary development tool for .NET applications.

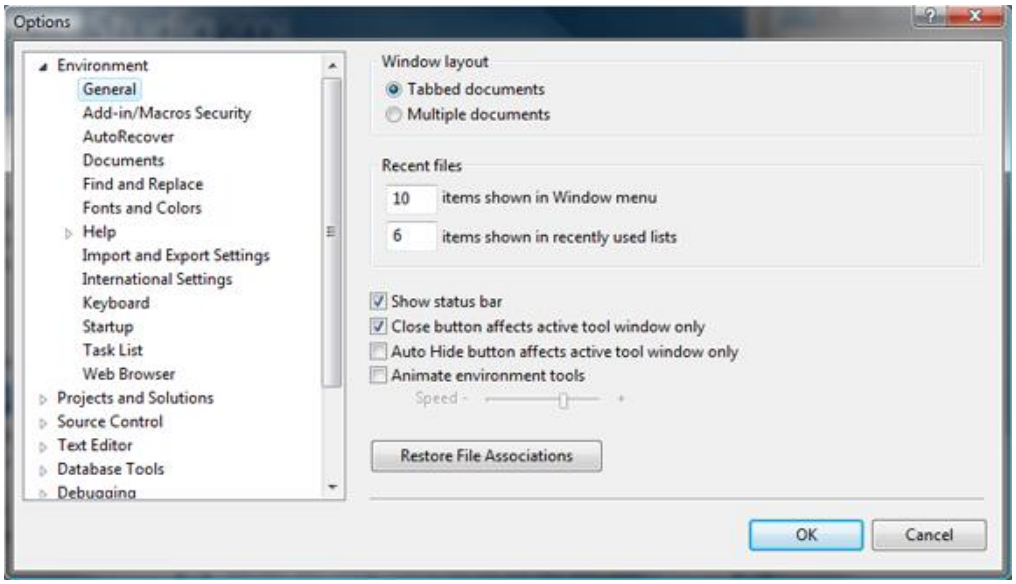
Start Page



When you start Visual Studio .NET you will see a "Start Page" like the one shown above. The links down the left side of the page lead you to online documentation and access to an online .NET community. The "My Profile" link gives options to make the IDE feel more familiar to programmers who are used to a particular environment. The Start Page may be enabled and disabled in the Options menu of Visual Studio.

Under Tools >Options [Environment>Startup]

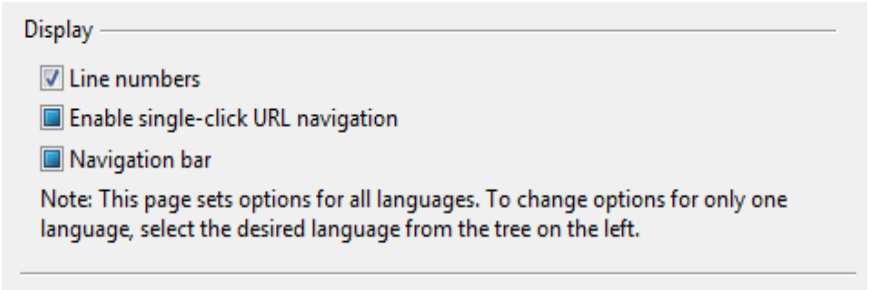
Options



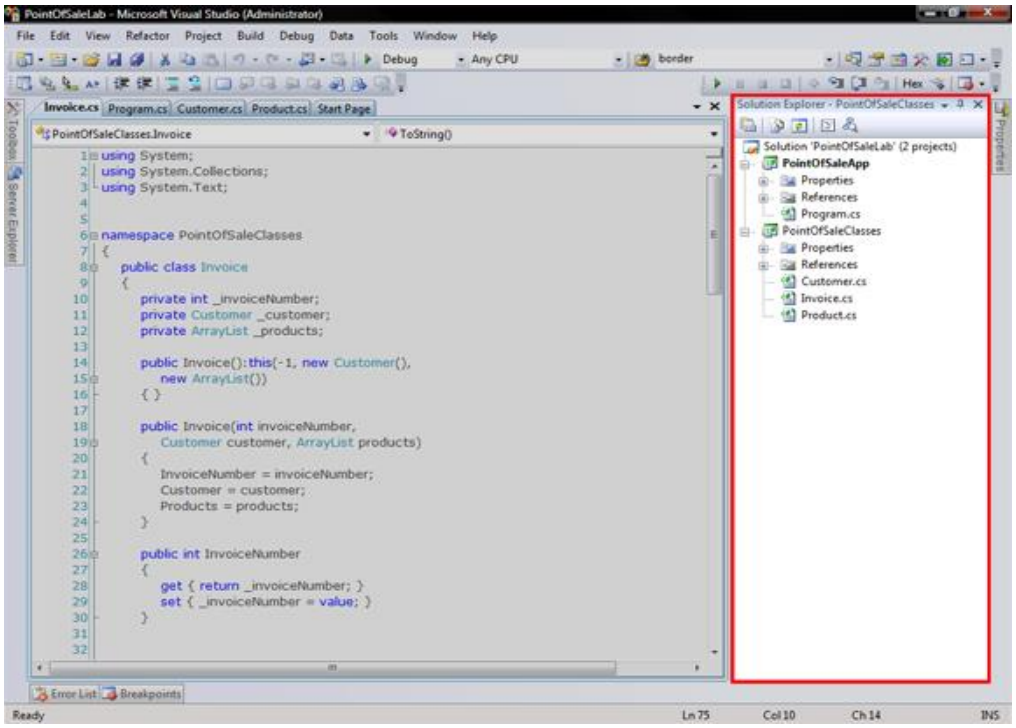
The Options dialogue may be accessed from the Tools menu. With this dialog box you may change fonts, color, line numbering, and how tabs are handled in each language you work with, default locations for files, assigning shortcut keys, and many other default settings.

Change the font used within Visual Studio under: [Environment>Fonts and Colors]

Turn on line numbers under: [Text Editor > All Languages]



Solution Explorer



The Solution Explorer provides a hierarchical view of the files, references, and other resources your project uses. You can think of a solution as a portfolio of projects.

The Solution Explorer can be launched from an icon on the standard toolbar.

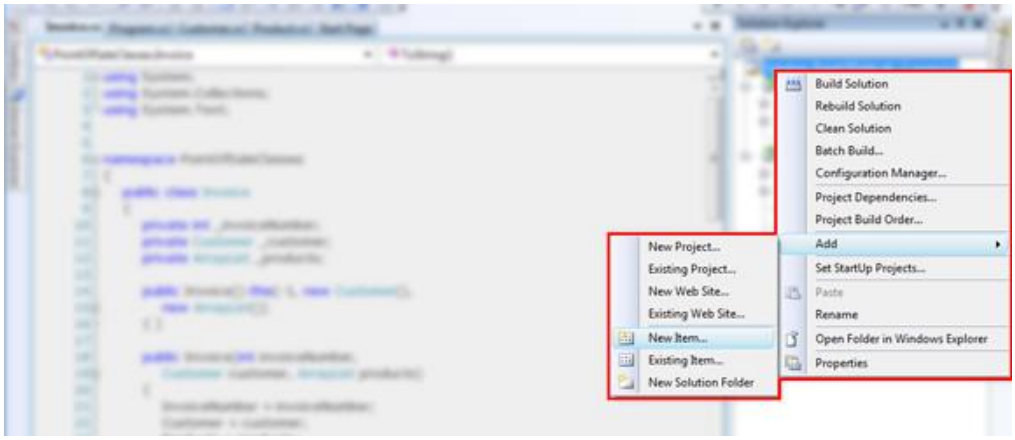
More IDE Components

1. Toolbox-the toolbox contains grouped components that can be used to customize applications. It is available on the standard toolbar.

2. Properties-the properties window allows you to see and change values that affect the look and feel of objects, it is available on the standard toolbar.

3. New Project Dialog Box-the new project dialog box can be viewed by selecting: File > New > Project. It allows for the creation of applications in a variety of languages.

4. Add New Item Dialog Box-the add new item dialog box can be viewed by first right-clicking on the solution within the Solution Explorer and within the contextual menu. Choose: Add > New Item.



In addition to the typical Help options in the Help menu of the standard toolbar, Dynamic Help is also available.

A Simple Program

There are ways to compile, debug and run projects in Visual Studio.

1. Build:
    a. CTRL + SHIFT + B (or F6)
    b. Select Build Solution from the Build menu.
    c. Display the Build Toolbar.
2. Debug:
    a. F5
    b. Select Start from the Debug menu.
    c. Display the Debug Toolbar.
3. Run:
    a. CTRL + F5
    b. Select Start without Debugging from the Debug menu.

Create, Build, and Run a Hello World program.

C# Language Fundamentals

Lesson Objectives

1. Comments
2. Intrinsic Data Types
3. Variables and Keywords
4. Operators
5. Strings
6. Arrays
7. Main Method
8. ReadLine and WriteLine Methods

Syntax Basics

1. White Space
2. Case sensitivity
3. Comments
4. Semi-Colons

Except in obvious places, white space is optional in its use. Good programming practices include the proper use of spacing and indenting to write readable and maintainable code.

Comments:

`//`	indicates a single line comment
`/* … */`	indicates multiple line comments
`///`	XML documentation can be created from the comments in our C# code

Data Types

1. Data types are either Value-Type or Reference-Type.
2. C# has several intrinsic data types.
3. Most intrinsic types map to System types of the Common Language Specification (CLS).

Whole number value-types

C# alias	CTS type	Size	Value range	Signed / Unsigned
sbtye	System.SByte	8 bits	-128 – 127	Signed
byte	System.Byte	8 bits	0 – 255	Unsigned
short	System.Int16	16 bits	-32,768 – 32,767	Signed
ushort	System.UInt16	16 bits	0 – 65,535	Unsigned
int	System.Int32	32 bits	-2,147,483,648 – 2,147,483,647	Signed
uint	System.UInt32	32 bits	0 – 4294967295	Unsigned
long	System.Int64	64 bits	-9,223,372,036,854,775,808 – 9,223,372,036,854,775,807	Signed
ulong	System.UInt64	64 bits	0 – 18,446,744,073,709,551,615	Unsigned

Floating point value-types

C# alias	CTS type	Size	Value range	Signed / Unsigned
float	System.Single	32 bits		Signed
double	System.Double	64 bits		Signed
Decimal	System.Decimal	128 bits		Signed

Other intrinsic data types

C# alias	CTS type	Size	Value range	Signed / Unsigned
char	System.Char	16 bits	Single character value	
bool	System.Boolean		true or false	
string	System.String			
object	System.Object			

Strings are C# intrinsic variables, but are of reference type.

Note that int (Int32) is the default whole number type, and double (Double) is the default floating point type. So whole number literals (like 137) will be initially assumed by the compiler to be ints, whereas floating point literals (like 17.9) will be seen as doubles. A shortcut to explicitly cast a floating point literal number (like 3.14) to a float is to add the F suffix (3.14F). Similarly, to explicitly cast the same literal to a decimal, add the M suffix (3.14M). Also, although the ranges are much larger for float and double, decimal is the most precise form of floating point data, so it should generally be used when multiplication and division are involved and precision is a priority over range.C# has four built in data types that do not match to the CLS: sbyte, ushort, uint, ulong. In other words, they cannot match other languages' data types.

Variables

1. Case sensitivity
2. Variable Naming Rules
3. Variable Naming Conventions

Rules

1. Start each variable with a letter or underscore (this convention will be discussed later).
2. After the first character use underscores, letters, or numbers.
3. Do not use one of the reserved keywords.
4. Names must be unique within scope.

Conventions

1. Avoid using all uppercase letters.
2. Avoid beginning with an underscore.
3. Avoid using abbreviations.
4. Use Camel or Pascal Casing naming in the multiple word names.
5. Avoid using keywords in a different case.
6. Use Hungarian notation to help identify the variable type.

C# Keywords			
abstract	event	new	struct
as	explicit	null	switch
base	extern	object	this
bool	false	operator	throw
break	finally	out	true
byte	fixed	override	try
case	float	params	typeof
catch	for	private	uint
char	foreach	protected	ulong
checked	goto	public	unchecked
class	if	readonly	unsafe
const	implicit	ref	ushort

These keywords do NOT need to be memorized.

Variable Declaration

1. Declaration Syntax
    a. Modifiers
    b. Type
    c. Name
    d. Assignment
2. Declaring multiple variables
3. Assigning values
4. Constants

Standard variable declaration:
<data type> <identifier>

```
int itemCount;
```

Declaring multiple variables of the same type in one statement:
<data type> <identifier>, <identifier>, <identifier>

```
int shippingNumber, employeeNumber;
```

Assign values to variables:
<identifier> = <value>

```
employeeNumber = 23;
```

Initialize variable when it is declared:
<data type> <identifier> = <value>

```
int numberOfBooks = 6
```

A constant (a variable that cannot be changed once initialized) is declared as follows:

```
const double POUNDS_PER_GALLONS = 8.33;
```

Converting Data Types

1. C# is strongly typed.
2. Converting between data types is also known as casting.
3. Two types of casting:
    a. Implicit
    b. Explicit

Implicit casting occurs automatically when moving to a larger compatible data type.

```
int x = 32;
long y = x;
```

This convention works fine because a long is larger than an int.

Explicit casting requires the use of a cast expression. The following example shows how to convert a long into an int.

```
long myLong = 8776;
int myInt = (int)myLong;
```

You must use M/m or F/f suffix when assigning decimal point values to decimal or float variables, respectively. No suffix is required when assigning to a double, as all decimal point literal values are interpreted as doubles by default.

```
decimal fraction1 = 1023.3m; //M or m suffix needed
float fraction2 = 345.97f; //F or f suffix needed
double fraction3 = 1023.3; //no suffix needed
decimal fraction4 = 200; //no suffix needed, but M or m wouldn't be wrong
```

Operators

- Arithmetic Operators
- Relational Operators
- Equality Operators
- Unary Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators

Arithmetic Operators:

Multiplication	*
Division	/
Modulus	%
Addition	+
Subtraction	-

Relational Operators:

Greater than	>
Less than	<
Greater than or equal to	>=
Less than or equal to	<=

Equality Operators:

Is equal to	==
Is not equal to	!=

Unary Operators:

Plus (positive)	+
Minus (negative)	-
Increment by one	++
Decrement by one	--

Logical Operators:

Conditional AND	&&
Conditional OR	||

Bitwise Operators:

Bitewise AND	&
Bitewise OR	|
Bitewise XOR	^
Bitewise Complement	~
Shift left	<<
Shift right	>>

Assignment Operators:

Assignment	=
Multiplication assignment	*=
Division assignment	/=
Modulus assignment	%=
Addition assignment	+=
Subtraction assignment	-=
Shift left assignment	<<=
Shift right assignment	>>=
Logical AND assignment	&=
Logical OR assignment	|=
Logical XOR assignment	^=

*Assignment operators are a combination of other operators and assignment. These provide programmers with handy shortcuts.

```
x = x+7;
```

is the same as

```
x +=7;
```

Operator Precedence

1. Using multiple operators in the same declaration will generate results based on the order of precedence of each operator.
2. Use parentheses to eliminate unwanted precedence concerns and to keep your code readable and maintainable.

The Main method

1. Program execution begins at the Main method.
2. There are a few different ways in which the Main method can be created.

A typical Main method would be:

```
public static void Main()
{
}
```

Another implementation of the Main method is:

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

Other implementations of the Main, and a discussion of its syntax will occur in subsequent chapters, for now it's worthwhile to understand that when a program is ran, execution begins at the Main method.

Strings

1. Strings are reference-type variables in C#.
2. The String class is part of the FCL (note case).
3. Declaring and Initializing

Even though the String class is part of the FCL, we can use it directly because it is intrinsic data in C#.

Use double quotes when declaring and initializing:

```
string name = "Bob";
```

The + operator is overloaded for string concatenation.

Strings are immutable.

Manipulating Strings

Common string methods:

- Substring()
- ToUpper()
- ToLower()
- Equals()
- Trim()
- Replace()

The escape character in C# is the backward slash (\) character in a format string to turn off the special meaning of the character that follows it. For example "\{" will cause the literal "{" to be displayed.

`\n` inserts a new line
`\t` inserts a horizontal tab into a string literal
`\\` inserts a backslash into a string literal
`\u` inserts a Unicode character into a string literal (requires Unicode # after \u).
Example: `\u00F0` for the degree ° symbol
`\"` inserts a double quote inside a string literal

Arrays

1. An array is a sequence of elements of the same type.
2. Declaring and Initializing
3. Manipulating Arrays

Declare and Initialize an array with a specified length:

```
int [] nbrs = new int [5];
```

Declare and Initialize an array with specified values:

```
int [] nbrs = {12,45,6,3};
```

Reassign individual indexes:

```
nbrs [0] = 23;
nbrs [1] = 4;
nbrs [2] = 597;
nbrs [3] = 88;
nbrs [4] = 100;
```

Access the length of an Array using the property length.

```
nbrs.Length
```

Arrays cannot be resized once initialized.

An array stores values on one data type.

Arrays can also be sorted, rearranging the values in a specific order from one index to the next:

```
Array.Sort(nbrs); //sorts the values in ascending order
Array.Reverse(nbrs); //literally reverses the order of the values
```

The code above demonstrates declaring and initializing build in arrays in .NET. There are other types of arrays or collections that will be covered in the advanced topics.

Console input and output

1. The Console class
2. Write() and WriteLine() methods
3. Read() and ReadLine() methods

The Console class is available in the System Namespace. As with all classes used from the FCL you must either fully qualify the Console class methods at each use, or add the 'using' directive to your code.

```
System.Console.WriteLine("Hello");
```

or

```
using System;
Console.WriteLine("Hello")'
```

The Read() and ReadLine() methods return an int and string respectively and must be cast or parsed respectively.

These methods are used strictly for console applications.

String Formatting

1. String formatting syntax
2. Numeric formatting

String formatting examples:

```
Console.WriteLine("The sum of {0} and {1} is {2}.",100,130,100+130);
```

or

```
int x = 100;
int y = 130;
Console.WriteLine("the sum of {0} and {1} is {2}.", x, y, x + y);
```

Compared to the same output using concatentation:

```
Console.WriteLine("The sum of " + x " and " + y + " is " + (x+y) + ".");
```

String formatting can be done with format specifiers for numeric types. Examples:

```
Console.WriteLine("I won {0:C} dollars", 88);
```

C Displays the number as currency using the local currency symbol and conventions.
N Displays the number with embedded commas.
X Displays the number in hexadecimal.

Format specifiers for string formatting can also be applied to DateTime objects:

```
Console.WriteLine("Today is {0:D}", DateTime.Today);
```

D Long date syntax (no time)
d Short date syntax (no time)
T Long time syntax (no date)
t Short time syntax (no date)
F Long, full datetime syntax
f Short, full datetime syntax

Reminders

1. Program execution begins at the Main method.
2. C# is case sensitive and strongly typed.
3. End all statements with a semi-colon (;).

Lab3AWater.cs

1. Write a program that will:
    a. Assume that a gallon of water weighs 8.33 pounds.
    b. Prompt the user to enter a number of gallons.
    c. Display the total weight of the water in pounds.

Lab3BChange.cs

2. Write a program that will:
    a. Prompt the user to enter an amount of dollars and cents (for example 1.18).
    b. Display the number of quarters, dimes, nickels, and pennies to make the amount.

Control Flow Statements

Lesson Objectives

1. If Statements
2. Switches
3. While
4. Do While
5. For
6. Foreach
7. Conditional Operator

if Statements

if Statement (also called conditional statement)

- if(<condition>)
- if(<condition>) else if (<condition>)
- if(<condition>) else
- if(<condition>) else if (<condition>) else

The `if` statement contains a test to determine whether a specific condition is true, and if so, the `if`'s code block runs. Otherwise (if the condition is false), the resulting code block does not run. Regardless, the processor moves on to any code that follows after the `if`'s code block is run or skipped.

`if` statements can be also be thought of as trees, with one or more branches. The `if` branch is always the first; any other branches are optional. However, no matter how many "branches" a tree has, no more than one of them will actually run its block of code. If you want to have multiple conditional blocks of code that are all evaluated *independently* of each other, those would be completely separate `if` statements (or trees).

But to design an `if` statement that runs just one block of code depending on the result of varying circumstances, you can add the optional `else if` or `else` clauses. For example, the originating `if` branch of an "if tree" can be followed by one or more `else if` statements, which would also have conditions they are testing against. You can think of `else if` branches as secondary `if` statements that are only evaluated if previous branches of the tree did not run their code blocks. So an `else if` will only be processed if the `if` or `else if` branches above it in this tree did not evaluate to true and therefore did not run. As soon as one of the branches of the tree evaluates to true, its code block fires and then the processor moves to the end of the tree to continue on with any remaining code. You can have as many `else if` branches in an `if` tree as needed.

Finally, an `if` tree can optionally end with an `else` branch. If included, this clause ends the `if` tree, regardless of whether there are any else if branches in the tree. The else clause does not have any condition it is testing for – it is the "all else failed" option that runs a block of code if all previous branches of the tree failed to run.

Scope

The { character begins a scope.
The } character ends a scope.
Scope is not required, however only a single statement will apply to the condition, and can cause compile errors in certain situations.

If - else if - else example:

```
int score = 90;
if(score >= 90 && score <=100)
{
      Console.WriteLine("You got an A");
}
else if(score >= 80 && score <=89)
{
      Console.WriteLine("You got a B");
}
else
{
      Console.WriteLine("You got an F");
}
```

switch Statement

switch Statements

- Used to select between an infinite number of cases.
- Switch on strings, whole number data types and chars.
- All cases must end with the break keyword.

switch example:

```
//receives a string from the user
string str = Console.Readline();

//converts (parses) string to an int
int nbr = int.Parse(str);

switch (nbr)
{
      case 1:
            Console.WriteLine("user typed in 1");
            break;
      case 2:
            Console.WriteLine("user typed in 2");
            break;
      default:
            Console.WriteLine("user typed in another value");
            break;
}
```

Loops

1. Determinate loops
    a. for loop
    b. foreach loop

2. Indeterminate loops
    a. while loops
    b. do while loop

for loop syntax:

```
for(<initialize> ; <condition> ; <update>)
{
}
```

foreach loop syntax:

```
foreach (<data type> <identifier> in <expression>)
{
}
```

while loops are indeterminate loops that execute 0 or more times.

do while loops are also indeterminate, and execute 1 or more times. Use these loops when the execution of behavior needs to occur at least once no matter what.

Scope is not required, but highly recommended for these structures.

while loop syntax:

```
while(<condition>)
{
}
```

do while loop syntax:

```
do
{
}
while (<condition>);
```

The conditional or ternary operator (?:)

```
<condition> ? <expression> : <expression>
```

ternary operator example:

```
int age = 17;
string drinkType = (age >= 21) ? “Adult Beverage” : “Kool-Aid”;
```

The ternary operator is a shortcut syntax for a simple if/else statement which uses ? and : symbols. It starts with a test, followed by a question mark (?), followed by the simplistic resulting code if the test is true, followed by a colon (:) and the code to run if the test is false. The ternary operator provides a very quick syntax but can be harder to read than a traditional `if` structure, and does not allow for the resulting code for true and false results to be longer than one simple statement. Because of this lack of flexibility, it is often recommended to use traditional `if` statements over a ternary operator.

The scopeless if

It is possible to write an if statement without the curly braces (or scope), as long as the resulting "code block" is only one line long. For the same reason as with the ternary operator above, it isn't recommended. It can be harder to read and due to the inflexibility of the syntax, it is certainly is not as maintainable.

scopeless if statement example:

```
if(age > 12)
Console.WriteLine("You are a teenager or older!");
```

Lab 4AGrades.cs

1. Write a program that will:
    a. Ask the user to enter a score between 0 and 100.
    b. Display the letter grade for the score, using the following logic:

90 +	A
89-80	B
79-70	C
69-60	D
59 and below	F

Lab 4BReverse.cs

2. Write a program that will:
    a. Accept 5 lines of user input, placing each line in a member of a string array.
    b. After the 5th line, the program should print the lines to console in reverse order (5th line should be the 1st line).

Structures

Lesson Objectives

1. What are structures?
2. How to declare structures.
3. How to create structures.
4. Accessing Structure Members
5. Enumerations

What are Structures?

- A structure is a grouping of arbitrary types.
- Structures are used for putting similar data together to create a new type.
- Structures are value types.

Structures example:

```
struct Time
{
      public int hour;
            public int minute;
            public int seconds;
}
```

Value types are declared on the stack, resulting in higher efficiency in many circumstances. Structures do not support inheritance (they are implicitly sealed).

Declaring Structures

- You must define what a structure is before you can use it.
- A structure can contain any type, including other structures.

Another structure example:

```
struct Employee
{
      public int empNbr;
      public string name;
      public double salary;
}
```

*Structures can also contain methods and properties, but traditionally structures often just have data.

Creating and Using

- Once a structure is declared, you can create one or more variables of that structure type.
- Access structure members using the dot operator.
- You must initialize a structure member before you can use it.

Examples:

```
Time lunch;
lunch.hour = 12;
lunch.minute = 0;
lunch.second = 0;
employee emp1;
emp1.empNbr = 101;
emp1.name = "Bob";
emp1.salary = 32000.00;
```

Structures can have constructors that initialize their data. We will discuss constructors when we introduce classes.

Enumerations

- Enumerations are a set of constant whole number variables.
- The variables in an enumeration are a related set of names.

Example:

```
enum Color {Red, Green, Blue};
Color c = Color.Blue;
```

In the above example, Red, Green, and Blue are represented in the computer as 0, 1, and 2 respectively.

Lab 5

1. Declare a structure called *Account* containing a string called *name* and a double value called *balance*.

2. Write a program that:
    a. Creates an Account structure.
    b. Ask the user for their name and initial balance, assign them to appropriate members of the structure.
    c. Prompt and allow the user to make a deposit, withdraw, initiate a balance inquiry, or exit the program.

Methods and Parameters

Lesson Objectives

1. Declaring Methods
2. Calling Methods
3. Using Parameters
4. The Return Statement
5. Method Signatures
6. Method Overloading
7. Advanced Parameter Concepts

Method Declaration

- A method is a group of C# statements that have been brought together and named.
- Methods must be declared in a class or struct.
- Creating methods
- Method syntax

When creating methods you must specify the following:

- Name (first letter uppercase by convention)
- Parameter list
- Body of the method

Method declaration syntax:

- Access Modifier
- Return type
- Name
- Parameter list
- Body

```
<access modifier> <return type> <identifier> ([param, param])
{
}
```

Method declaration example:

```
public void MyMethod ()
}
{
```

Calling Method

You can call methods…

- From within the same class.
- From within another class.
- From inside another method.

How you access a method depends on where the call is made in relation to where the method is defined. The explanation of the necessary elements of the method calls may be unclear until the understanding of Classes and Objects is achieved.

From within the same class:

- Use the method's name followed by a parameter list in parentheses.

From within another class:

- You must indicate to the compiler which class contains the method to call. The called method must be declared with the **public** keyword, allowing the method to be accessible outside of the class.

From inside another method:

- One method can call another method, which can call other methods, and so on.

Methods Parameters

- Declaring parameters – When declaring place parameters between the parentheses after the method name. Define type and name for each parameter. Multiple parameters are separated by commas.

- Parameters are used to pass values into methods.

Method declared with parameters example:

```
public void MyMethod (int age, string name)
{
      //code
}
```

- Calling parameters – When calling a method that takes parameters, supply a value of the proper type for each method.

Method call with values passed to the parameters example:

```
MyMethod (17, "Ferris Bueler");
```

Return

- Return statement normally used when method return type is something other than void.
- When the return statement in a method is reached, program flow returns to the caller of the method.
- The return data must be compatible with the return type.

```
public void DisplayIncome(double dbl)
{
      Console.WriteLine(dbl);
      //no return is necessary because the return type is void
}

public double ComputeTax(double income)
{
      double tax = income * /99;
      return tax;
      //when a method's return type is NOT void you must return a value
      //of that type with the return keyword.
}
```

Signatures and Overloading

- A method signature consists of the method's name, the number, type and order of the parameters.
- You cannot have two methods with the same signature together. (compiler error)
- Method overloading is creating method with the same name but different signatures.

```
public double ComputeTax(double income)
{
      double tax = income * .99;
      return tax;
}

public double ComputeTax(double income, double taxRate)
{
      double tax = income * taxRate;
      return tax;
}
```

Advanced Parameter Concepts

1. Pass by Value
    a. Default mechanism for passing parameters.
    b. Variable can be charged inside the method with no effect on the value outside the method.

2. Pass by reference
    a. Use the **ref** keyword in the method declaration of the call.
    b. Variable changed inside a method will be seen outside the method.

3. Output parameters
    a. Like ref, but values are not passed into the method.
    b. Use the **out** keyword in method declaration of the call.
    c. Used to retrieve more than one value from a method, where the return would be insufficient (since there can only be one return type).

4. Variable length parameters
    a. Use the **params** keyword.
    b. Declare only on params per method.
    c. Place parameter at the end of the parameter list.
    d. Declare the parameter as a single dimension array type.
    e. Use **params** keyword only in the declaration, not in the call.

The following examples show how to declare a variable length parameter list:

```
public void DisplayArrayOfInts(string msg, params int [] list)
{
      Console.WriteLine(msg);
      Foreach(int I in list)
      {
            Console.WriteLine(i);
      }
}
```

To call this method:

```
DisplayArrayOfInts("Here they are!", new int[3] {3,45,67});
```

or

```
DisplayArrayOfInts("Here they are!", 3,45,76);
```

Lab 6

1. Declare a structure called *Account* containing a string called *name* and a double value called *balance*.

2. Add a method called *Display* which will display a balance to the console.

3. Add a method called *MonthlyInterest* that takes a double for the annual interest rate. The method should calculate the interest for a single month.

4. Write a program that creates an *Account* structure. Ask the user for their name and initial balance and assign them to appropriate members of the structure.

5. In this program, prompt and allow the user to make a deposit, withdraw, initiate a balance inquiry, and display monthly interest or exit the program.

Classes and Objects

Lesson Objectives

1. Object-Oriented Programming
2. Attributes (member variables)
3. Constructors
4. Properties
5. ToString() method
6. Readonly

Object-Oriented Concepts

1. What is a class?
    - A class is a named syntactic that describes common behavior and attributes.
    - Classes are like blueprints for a house.
    - Classes are used to represent real-world objects in programming.
    - Classes contain members. (attributes, methods, properties, etc.)

2. What is an object?
    - An object is an instance of a class.
    - The house (instance) is built from the blueprint (class).
    - One blueprint (class) can be used to make many houses (instances).

3. Encapsulation: The concept of hiding details about a specific thing or object. (Most people know that pushing the gas pedal down on a car makes the car go, however few people know exactly how the engine works to achieve this, in programming we encapsulate details about our objects in much the same way).

4. Identity: An object's identity is that which makes it distinguishable from all other objects of the same class.

5. Behavior: An object's behavior consists of the tasks that it can perform. In our classes, behavior is defined by the methods in a class.

6. State: Is the current value of the attributes of an object.

Attributes (Fields)

- Attributes (fields) are also known as a class member.
- Attributes should be marked as private.
- Attributes can be declared anywhere in a class as long as they are declared at class level scope.

Create a simple class example:

```
class Employee
{
	//attributes/fields
	public int empNbr;
	public string fname;
	public string lname;
}
```

Constructors

- Used to create an instance of a class.
- Named the same as the class name.
- No return type.
- Defines initial value of a class' attributes (data members).
- A default constructor will be provided by the compiler in some circumstances.
- Constructors can be overloaded.

Attributes get an initial value even if they are not initialized explicitly in a constructor.

Continued simple class example:

```
class Employee
{
	//attributes/fields
	public int empNbr;
	public string fname;
	public string lname;

	//ctor
	public Employee(int empNumber, string firstName, string lastName)
	{
		this.empNbr = empNumber;
		this.fname = firstName;
		this.lname = lastName;
	}
}
```

Calling a constructor

- A constructor is called when a class is instantiated.
- The **new** keyword.

Continued simple class example:

```
Employee emp1 = new Employee(1001, "Fred", "Derf");
```

Properties

- Access data members (attributes).
- Modify data members (attributes).
- Use **get**, **return**, and **set**, **value** keywords.

Properties are designed to allow controlled access to member data (attributes) of a class. Specifically they control the ability to view and modify attributes values.

```
class Time
{
      //attributes/fields
      private int _hour;

      //property encapsulates private attribute _hour;
      {
            get {return _hour;}
            set {hour=value;}
      }
}

//instantiate a new time object
Time t = new Time ();

//set (assign) a new value
t.Hour = 1;

//get (read) the value
int x = t.Hour;
```

ToString()

- A string representation of an object.

```
public override string Tostring()
{
      return "";
}
```

Static

Static Variables: Shared by all instances of a class, therefore a change to a static variable will be reflected in all instances.

Static Methods: Cannot access non-static members. They also may not use the **this** reference.

Some Math class methods:

- Abs(x)
- Ceiling(x)
- Floor(x)
- Max(x)
- Min(x)
- Pow(x)

Readonly

- **readonly** and **const** are very similar.
- Both create constant data.

const variables must be initialized when declared.

readonly variables may be initialized when declared or in the constructors of their class.

If a **readonly** variable is not initialized, it will be given a default value.

Structs vs. Classes

Struct	Class
• A struct is a blueprint for a value. • No identity, accessible state, no added behavior.	• A class is a blueprint for an object. • Identity, controlled access to state, added behavior.

*Variables of the struct type can contain methods, however it is recommended that they do not.

Lab 7

1. Create a standard class called *Account* with the following members:

    a. 2 attributes
        - String name
        - Double balance

    b. 1 static attribute
        - Double rate

    c. A 0 argument constructor and a 2 argument constructor.

    d. Properties for all 3 attributes.

    e. An appropriate ToString method.

    f. Other methods that perform the following functions:
        - Deposit
        - Withdrawal
        - Calculate monthly interest

    g. Create a second class to test the account class:
        - Set the interest rate.
        - Get the users account name and initial account balance.
        - Display a menu for the user to choose various account options.

Inheritance and Polymorphism

Lesson Objectives

1. Base/Derived Classes
2. Virtual Methods
3. Accessing Base Class Members
4. Late Binding
5. New To Hide Methods
6. Abstract
7. Interfaces
8. Sealed Classes

What is Inheritance?

- Is-a-kind-of
- Maximizes code re-use
- Allows for specialization

Inheritance is a form of software reusability in which classes are created by absorbing an existing class's data and behavior, and adding new capabilities to them.

The .NET class System.Object is the base class for all classes in the FCL. C#'s intrinsic object class maps directly to System.Object, so it is sufficient to say that the object class in C# is the base class of all classes.

All classes will inherit from object even if the relationship is not specifically stated. The object class contains several methods, including the ToString().

Derived and Base Classes

- Syntax for deriving from a base class.
- Inheritance of class members.
- Derived classes cannot have a greater access level than its base.
- System.Object

Syntax:

```
class Animal
{
}

class Dog:Animal
{
}
```

A derived class inherits everything from its base class except constructors and destructors. These can be accessed through special syntax discussed later in the lesson.

Base Class Members

- Protected access modifier
- Calling base class constructors
- Implicit base constructor calls
- The **base** keyword

The **protected** keyword acts like the **private** keyword to derived classes. Inherited members that are protected in the base class are automatically protected in the derived class. Protected cannot be used in structs.

Base class constructor call syntax:

```
class Animal
{
      protected string_typeOfSound;

      //ctor
      public Animal(string typeOfSound)
      {
            _typeOfSound=typeOfSound;
      }
}

class Dog:Animal
{
      //ctor
      public Dog(string typeOfSound):base(typeOfSound)
      {
      }
}
```

The base keyword is also used with a .(dot) to indentify base class member.

Polymorphic Behavior

- Virtual Methods
- Override

Polymorphism allows one method call to perform different actions, depending on the type of object receiving the call.

Declaring a method to be virtual means that the method can be overridden in a derived class.

```
public virtual string MyMethod()
{
}
```

You can override identical virtual methods from base classes to create polymorphic behavior.

Rules

1. Virtual methods cannot be static.
2. Virtual methods cannot be private.
3. Virtual methods must have a method body.
4. Override must match an existing virtual method.
5. You can override an override method.
6. Override methods cannot also be virtual.
7. Override methods cannot be static or private.
8. An override method must have the same access level, return type, name, and parameters as the virtual method. (They must be identical).

Late Binding

- The decision of which method to use in a polymorphic chain is made at run time.
- Derived class instances can be stored in base class variables.

When a derived class instance is stored in a base call variable, the compiler only sees it as an instance of the base class. Therefore, any method call on that instance must be supported by the base class. At runtime, the CLR will determine if the derived class has its own version of the method invoked (polymorphism).

New

- Add to non-override methods to resolve name clashes.
- Add to virtual methods of the same name in derived classes.

Sealed Classes

- Prevents inheritance.
- Used to optimize runtime performance.

Syntax example:

```
public sealed class MyClass
{
}
```

Abstract Classes

- Designed to force inheritance.
- May contain abstract methods or concrete (non-abstract) methods.
- If a class contains an abstract method, it too must be abstract.
- An abstract class cannot be instantiated.

Syntax example:

```
//abstract class
Abstract class MyClass
{

}
```

Abstract Methods

- Must be inside an abstract class.
- Contain no method body.
- Are virtual by nature (cannot mark them as virtual).
- Can override base class virtual and override methods.

Syntax example continued:

```
//abstract class

  abstract class MyClass
  {
      //abstract method
      public abstract void MyMethod();
  }
```

Interfaces

- Can be thought of as a type of class that is abstract by nature.
- Convention puts an "I" before interface name.
- Classes implement interfaces.
- Interfaces extend other interfaces.
- Interface methods are abstract by nature and can have no method body.
- Classes must implement all interface methods.

Interfaces are C#'s way of achieving multiple inheritance, and not suffering the confusion that a diamond inheritance model can bring. Think of an interface as a binding contract between class and interface. The class gets the functionality of the interface, but agrees to implement all of the methods in return.

You cannot use the abstract keyword when defining an interface or its methods.

Classes can implement multiple interfaces, using the same syntax as it does for extending a base class, each interface listed is separated by commas.

Lab 8

1. Create a class called *Payment.*
    a. Double amount
    b. 2 ctors 0/1

2. Create a class called *CheckPayment* extending *Payment.*
    a. int checkNumber
    b. 2 ctors 0/2

3. Create a class called *CreditCardPayment* extending *Payment.*
    a. String C*reditCardNumber*
    b. 2 ctors 0/2

4. Create an app class.
    a. Main
    b. Prompt user to choose payment type.
    c. Process payment of that type.
    d. Print out payment information for that transaction.

Advanced Language Concepts

Lesson Objectives

1. “is” and “as” Operators
2. Operator Overloading
3. Boxing and Unboxing
4. Garbage Collection and Destructors
5. Aggregation and Composition

“is” and “as”

- The **is** operator checks to see if an object is of a particular type of runtime and returns a bool value.
- The **as** operator returns a casted object if true and null if false.

is syntax example:

```
if (obj is Circle)
{
      {(Circle)obj).bounce();
}
```

as syntax emample:

```
Circle c = obj as Circle;
if(c != null)
{
      c.bounce();
}
```

Boxing and Unboxing

- Boxing automatically creates reference-types out of value-type data.
- Boxing occurs when assigning a value-type to a reference type variable.
- Unboxing is done with an explicit cast, but must be the exact data type previously boxed.

```
//declare an int variable
int x = 3;

//declare an object variable
object obj;

//boxing
//place the int variable into the object variable
obj = x;

//declare another int variable
int y = 0;

//unbox
//cast the int variable inside the object variable back into an int
y = (int)ojb;
```

Boxing is particularly useful when using the collection classes.

Access Modifiers

- Access modifiers are keywords used to specify the declared accessibility of a member or a type.
- There are five access modifiers:
    1. public
    2. private
    3. protected
    4. internal
    5. protected internal

These modifiers make up five access levels:

Public: Indicates the method is freely accessible inside and outside of the class in which it is defined.

Private: Methods are only accessible in the class in which they are defined.

Protected: The method is accessible in the type in which it is defined, and in derived types of that type. This is used to give derived classes access to the methods in their base classes.

Internal: The method is only accessible to types defined in the same assembly.

Protected Internal: The method is accessible to types defined in the same assembly or to types in a derived assembly.

Garbage Collection

- Automatic progress of freeing memory where an object resides that is no longer strongly referred to.
- Reference-type variables are garbage collected when they have gone out of scope, lost a reference, or are set to null.
- Destructors are called prior to garbage collection.

The garbage collector can be called by using:

```
System.GC.Collect();
```

Destructor syntax:

```
Class Employee
{
      ~Employee()
      {
      }
}
```

Aggregation and Composition

- Complex objects built from simpler objects.
- Aggregation occurs when a class has other reference-type data as an attribute.
- Composition occurs when a class has value-type data as attributes, or when reference-type data is used as attributes and has no other purpose than to serve that class.

Exceptions

Lesson Objectives

1. To understand exceptions and the exception hierarchy.
2. To use try blocks to delimit code in which exceptions may occur.
3. To use catch blocks to handle exceptions.
4. To use finally blocks to release resources.
5. To throw your own exceptions.

Exception Hierarchy

- All exceptions derive from a class Exception, which is available in the System namespace.
- Exceptions are broken into two major categories.
- If a method throws an exception, we are not forced to hand that exception.

Exceptions are divided into two classes: SystemException and ApplicationException. The idea was that all exceptions in the FCL would extend from SystemException, and programmer defined exceptions would extend from ApplicationException. This did not happen.

Some languages force the handling of some exceptions. That is not the case in C#.

Handling Exceptions

- Methods that throw exceptions can be accessed inside of try blocks.
- If an exception occurs inside a try block, the block expires and program control is transferred.
- As with any other block of code, when a try block terminates, local variables defined in the try go out of scope.

Syntax example:

```
try
{
      //code below could possibly throw an exception
      myVariable.MethodCall();
}
```

Catch Blocks

- A try block can have zero to many catch blocks associated with it.
- Multiple catch blocks must be listed in their object hierarchy from most specific to least specific.
- A general catch block may be used once per try block.

If a try block does not have at least one catch block associated with it, it must have a finally block. You must catch exceptions from most specific to least, or your code will not compile. If a general catch block is used, it must be the last catch in the structure.

```
try
{
      //code that could possibly throw an exception
}
//catch more specific exception
catch (OutOfMemoryException caught)
{


}
//catch least specific exception
catch(Exception e)
{

}
```

Finally Block

All statements in a finally block are always executed, regardless of the course of control flow.

```
try
{
}
catch
{
}
finally
{
}
```

Raising Your Own Exceptions

- Throw the appropriate exception adding in a message string.
- You may throw system defined exceptions or exceptions that you have created.

Exception throwing syntax:

```
public void MyMethod()
{
      if ( x > 100 )
      {
            throw new Exception("Number is too high!");
      }
}
```

If you define your own exception, you must derive (directly or indirectly) from the Exception class.

Creating Custom Exceptions

- You handle application specific errors by creating user-defined exception classes.
- User-defined exception classes should extend ApplicationException.
- User-defined exceptions can be handled like any other .NET exception.

Syntax example:

```
using System;
public class MyCustomException:ApplicationException
{
      public MyCustomException(string msg):base(msg)
      {
      }
      public MyCustomException
      (string msg, Exception innerException):base(msg,innerException)
      {
      }
}
```

Lab 10

1. Add to our account class solution from Lab 7, by creating an OverdraftException classes.

2. Have the Withdrawal method thrown an OverdraftException if there is an attempt to withdrawal an amount greater than the balance.

3. Handle that exception in the application class.

Collections

Lesson Objectives

1. Identify the hierarchy of the FCL that contains collection classes.
2. Be able to identify several collection classes and their general behavior.
3. Create and manipulate ArrayLists and Hash Tables.

Hierarchy

- Collections are data types designed to hold other data.
- Collections are found in the System.Collections or the System.Collections.Specialized namespaces.
- Collections hold data of type object.

Any value-type data will be boxed automatically when inserted into a collection.

Some of the collections provided by .NET are called:

1. ArrayList()
2. Stack
3. Queue
4. Hashtable
5. SortedList

ArrayList

- Similar to an array, but can expand and contract.
- Increases in size automatically.
- Can contain objects of any type.

Declaration syntax example:

```
//instantiating an ArrayList object
ArrayList al = new ArrayList ();

//adds the string to the last index of the ArrayList
al.Add("Hello");
```

Properties:

1. Capacity
2. Count

Methods:

1. Add()
2. Clear()
3. Contains()
4. Insert()
5. Sort()
6. TrimToSize()

Hashtable

- Stores values in key-value pairings.
- Retrieve values using keys.
- Iterate through a Hashtable using an enumerator or a foreach statement.

Declare a Hashtable and add values:

```
Hashtable h = new Hashtable();
h.Add("key1", "value1");
h.Add("key2", "value2");
h.Add("key3", "value3");
```

Properties:

1. Count

Methods:

1. Add()
2. ContainsKey()
3. ContainsValue()
4. GetEnumerator()
5. Clear()

Stack and Queue Classes

- Stack class is a last in first out collection (lifo).
- Queue class is a first in first out collection (fifo).
- Both classes can be enumerated through.

Stack Methods:

1. Clear()
2. Peek()
3. Pop()
4. Push()
5. GetEnumerator()

Queue Methods:

1. Clear()
2. Peek()
3. Dequeue()
4. Enqueue()
5. GetEnumberator()

Lab 11

1. Write a standard book class with *isbn, title, author*, and *price* as attributes.

2. Write an app class that allows the user to do the following:
    a. Add books
    b. Remove books
    c. Display books
    d. Exit

3. Store book values in a Hashtable.

Solutions

Lab3AWater.cs Solution

```
// Lab3AWater.cs
using System;

namespace lab3
{
    public class Water
    {
        public static void Main()
        {
            const double PoundsPerGallon = 8.33;
            // Prompt the user to enter a number of gallons
            Console.Write("Please enter a number of gallons: ");
            string line = Console.ReadLine();
            double gallons = double.Parse(line);
            Console.WriteLine(
"{0} gallons of water weighs {1} pounds."
, gallons, gallons * PoundsPerGallon);
        }
    }
}
```

Lab3BChange.cs Solution

```
// Lab3Change.cs
using System;

namespace lab3
{
    public class CountChange
    {
        public static void Main()
        {
            // Prompt the user to enter a dollar and cent amount
            Console.Write("Please enter the amount of change to count: ");
            string line = Console.ReadLine();
            double amount = double.Parse(line);

            // Multiply amount by 100 and store as an int.
            // This will prevent rounding errors due to inexact
            // representations of number by double values
            int amountInCents = (int)(amount * 100);

            // compute quarters
            int quarters = amountInCents / 25;
            amountInCents %= 25;

            // compute dimes
            int dimes = amountInCents / 10;
            amountInCents %= 10;

            // compute nickels
            int nickels = amountInCents / 5;
            amountInCents %= 5;

            // pennies are what is left
            int pennies = amountInCents;
            Console.WriteLine(
                "{0} quarters\n{1} dimes\n{2} nickels\n{3} pennies"
                , quarters, dimes, nickels, pennies);
        }
    }
}
```

Lab4AGrades.cs Solution

```
// Lab4AGrades.cs
using System;

namespace lab4
{
    public class Grades
    {
        public static void Main()
        {
            // Prompt the user to enter a score
            Console.Write("Please enter a score (-1 to exit): ");
            string line = Console.ReadLine();
            int score = int.Parse(line);

            while (score != -1)
            {
                if (score >= 90)
                {
                    Console.WriteLine("A");
                }
                else if (score >= 80)
                {
                    Console.WriteLine("B");
                }
                else if (score >= 70)
                {
                    Console.WriteLine("C");
                }
                else if (score >= 60)
                {
                    Console.WriteLine("D");
                }
                else
                {
                    Console.WriteLine("F");
                }
                Console.Write("Please enter a score (-1 to exit): ");
                line = Console.ReadLine();
                score = int.Parse(line);
            }
        }
    }
}
```

Lab4BReverse.cs Solution

```
// Lab4BReverse.cs
using System;

namespace lab4
{
    public class Lines
    {
        public static void Main()
        {
            string[] lines = new string[5];
            // Ask the user to enter 5 lines of text
            for (int i = 0; i < lines.Length; i++)
            {
                Console.Write("Enter a line {0}: ", i + 1);
                lines[i] = Console.ReadLine();
            }
            Console.WriteLine();
            // Display the lines of text to the user in reverse order
            for (int i = lines.Length - 1; i >= 0; i--)
            {
                Console.WriteLine("Line {0}: {1}", i + 1, lines[i]);
            }
        }
    }
}
```

Lab 5 Solution

```
// Lab 5
// Account.cs
using System;
namespace lab5
{
    public enum AccountType
    {
        checking, savings
    }
    public struct Account
    {
        public string Name; public double Balance; public AccountType Type;
    }
    public class AccountApp
    {
        public static void Main()
        {
            Account myAccount;

            // Ask user for their name
            Console.Write("Enter your name: ");
            myAccount.Name = Console.ReadLine();

            // Ask user for their initial balance
            Console.Write("Enter your initial balance: ");
            myAccount.Balance = double.Parse(Console.ReadLine());

            // Ask user for the account type
            Console.Write("Enter the account type (checking or savings): ");
            myAccount.Type =
                (AccountType)Enum.Parse(typeof(AccountType),
Console.ReadLine());
```

Lab 5 Solution (Continued)

```
int choice;

            do
            {
                Console.WriteLine();
                Console.WriteLine("1) Deposit");
                Console.WriteLine("2) Withdraw");
                Console.WriteLine("3) Balance Inquiry");
                Console.WriteLine("4) Exit");
                Console.Write("Enter choice: ");
                choice = int.Parse(Console.ReadLine());
                while (choice < 1 || choice > 4)
                {
                    Console.Write(
"Invalid choice. Please enter valid choice: ");
                    choice = int.Parse(Console.ReadLine());
                }
                if (choice == 1)
                {
                    Console.Write("Enter amount to deposit: ");
                    double amount = double.Parse(Console.ReadLine());
                    myAccount.Balance += amount;
                }
                else if (choice == 2)
                {
                    Console.Write("Enter amount to withdraw: ");
                    double amount = double.Parse(Console.ReadLine());
                    myAccount.Balance -= amount;
                }
                else if (choice == 3)
                {
                    Console.WriteLine("{0}'s {2} account balance is {1:C}",
                        myAccount.Name, myAccount.Balance, myAccount.Type);
                }
            } while (choice != 4);
        }
    }
}
```

Lab 6 Solution

```
// Lab 6
// Account.cs
using System;
namespace lab6
{
    public struct Account
    {
        public string Name;
        public double Balance;

        public void Display()
        {
            Console.WriteLine(
                "{0}'s account balance is {1:C}", Name, Balance);
        }
        public void MonthlyInterest(double rate)
        {
            Balance += Balance * (rate / 12);
        }
    }
    public class AccountApp
    {
        public static void Main()
        {
            const double Rate = 0.05; Account myAccount;
            // Ask user for their name
            Console.Write("Enter your name: ");
            myAccount.Name = Console.ReadLine();
            // Ask user for their initial balance
            Console.Write("Enter your initial balance: ");
            myAccount.Balance = double.Parse(Console.ReadLine());
            int choice; do
            {
                Console.WriteLine();
                Console.WriteLine("1) Deposit");
                Console.WriteLine("2) Withdraw");
                Console.WriteLine("3) Balance Inquiry");
                Console.WriteLine("4) Compute Monthly Interest");
                Console.WriteLine("5) Exit");
                Console.Write("Enter choice: ");
                choice = int.Parse(Console.ReadLine());
```

Lab 6 Solution (Continued)

```
while (choice < 1 || choice > 5)
                {
                    Console.Write(
"Invalid choice. Please enter valid choice: ");
                    choice = int.Parse(Console.ReadLine());
                }
                if (choice == 1)
                {
                    Console.Write("Enter amount to deposit: ");
                    double amount = double.Parse(Console.ReadLine());
                    myAccount.Balance += amount;
                    myAccount.Display();
                }
                else if (choice == 2)
                {
                    Console.Write("Enter amount to withdraw: ");
                    double amount = double.Parse(Console.ReadLine());
                    myAccount.Balance -= amount;
                    myAccount.Display();
                }
                else if (choice == 3)
                {
                    myAccount.Display();
                }
                else if (choice == 4)
                {
                    myAccount.MonthlyInterest(Rate);
                    myAccount.Display();
                }
            } while (choice != 5);
        }
    }
}
```

Lab 7 Solution

```
// Lab 7
// Account.cs
using System;
namespace lab7
{
    public class Account
    {
        // attributes
        private string name;
        private double balance;
        // interest rate for accounts
        private static double rate;
        // constructors
        public Account() : this("", 0.0) { }
        public Account(string name, double balance)
        {
            Name = name;
            Balance = balance;
        }
        // properties
        public String Name
        {
            get { return name; }
            set { name = value; }
        }
        public double Balance
        {
            get { return balance; }
            set { balance = value; }
        }
        public static double Rate
        {
            get { return rate; }
            set { rate = value; }
        }
        // ToString
        public override string ToString()
        {
            return String.Format(
"Name is {0}, Balance is {1:C}", Name, Balance);
        }
```

Lab 7 Solution (Continued)

```
// Business methods
        public void Deposit(double amount)
        {
            Balance += amount;
        }
        public void Withdraw(double amount)
        {
            Balance -= amount;
        }
        public void MonthlyInterest()
        {
            Balance += Balance * (Rate / 12);
        }
    } // end of class Account
    public class AccountApp
    {
        public static void Main()
        {
            // Set the interest rate
            Account.Rate = 0.05;
            // Ask user for their name
            Console.Write("Enter your name: ");
            string name = Console.ReadLine();
            // Ask user for their initial balance
            Console.Write("Enter your initial balance: ");
            double balance = double.Parse(Console.ReadLine());
            Account myAccount = new Account(name, balance);
            int choice;
            do
            {
                Console.WriteLine();
                Console.WriteLine("1) Deposit");
                Console.WriteLine("2) Withdraw");
                Console.WriteLine("3) Balance Inquiry");
                Console.WriteLine("4) Compute Monthly Interest");
                Console.WriteLine("5) Exit");
                Console.Write("Enter choice: ");
                choice = int.Parse(Console.ReadLine());
                while (choice < 1 || choice > 5)
                {
                    Console.Write(
"Invalid choice. Please enter valid choice: ");
                    choice = int.Parse(Console.ReadLine());
                }
```

Lab 7 Solution (Continued)

```
if (choice == 1)
                {
                    Console.Write("Enter amount to deposit: ");
                    double amount = double.Parse(Console.ReadLine());
                    myAccount.Deposit(amount);
                    Console.WriteLine(myAccount);
                }
                else if (choice == 2)
                {
                    Console.Write("Enter amount to withdraw: ");
                    double amount = double.Parse(Console.ReadLine());
                    myAccount.Withdraw(amount);
                    Console.WriteLine(myAccount);
                }
                else if (choice == 3)
                {
                    Console.WriteLine(myAccount);
                }
                else if (choice == 4)
                {
                    myAccount.MonthlyInterest();
                    Console.WriteLine(myAccount);
                }
            }
            while (choice != 5);
        }
    } // end of class AccountApp
}
```

Lab 8 Solution

```
// Lab 8
// Payment.cs
using System;
namespace lab8
{
    public class Payment
    {
        // attributes
        private double amount;
        // constructors
        public Payment()
            : this(0.0)
        {
        }
        public Payment(double amount)
        {
            Amount = amount;
        }
        // properties
        public double Amount
        {
            get { return amount; }
            set { amount = value; }
        }
        // ToString
        public override string ToString()
        {
            return string.Format("Amount is {0}", Amount);
        }
    } // end of class Payment
```

Lab 8 Solution (Continued)

```
public class CheckPayment : Payment
    {
        // attributes
        private int checkNumber;
        // constructors
        public CheckPayment()
            : base()
        {
            CheckNumber = 100;
        }
        public CheckPayment(double amount, int checkNumber)
            : base(amount)
        {
            CheckNumber = checkNumber;
        }
        // properties
        public int CheckNumber
        {
            get { return checkNumber; }
            set { checkNumber = value; }
        }
        // ToString
        public override string ToString()
        {
            return base.ToString() + String.Format(
", Check number is {0}.", CheckNumber);
        }
    } // end of class CheckPayment
```

Lab 8 Solution (Continued)

```
public class CreditCardPayment : Payment
    {
        // attributes
        private string creditCardNumber;
        // constructors
        public CreditCardPayment()
            : base()
        {
            CreditCardNumber = "";
        }
        public CreditCardPayment(double amount, string creditCardNumber)
            : base(amount)
        {
            CreditCardNumber = creditCardNumber;
        }
        // properties
        public string CreditCardNumber
        {
            get { return creditCardNumber; }
            set { creditCardNumber = value; }
        }
        // ToString
        public override string ToString()
        {
            return base.ToString() + String.Format(
", Credit card number is {0}.", CreditCardNumber);
        }
    } // end of class CreditCardPayment
```

Lab 8 Solution (Continued)

```
public class PaymentApp
    {
        public static void Main()
        {
            Console.Write("Enter number of payments to process: ");
            int count = int.Parse(Console.ReadLine());

            Payment[] payments = new Payment[count];

            // ask user for information about each payment

            for (int i = 0; i < payments.Length; i++)
            {
                // List kind of payments
                Console.WriteLine("1) Cash Payment");
                Console.WriteLine("2) Check Payment");
                Console.WriteLine("3) Credit Card Payment");
                Console.WriteLine();

                // Ask user to enter choice
                Console.Write("Enter choice: ");
                int choice = int.Parse(Console.ReadLine());
                // validate choice
                while (choice < 1 || choice > 3)
                {
                    Console.Write("Invalid choice. Enter choice: ");
                    choice = int.Parse(Console.ReadLine());
                }
                // Get the amount from the user
                Console.Write("Enter payment amount: ");
                double amount = double.Parse(Console.ReadLine());
                if (choice == 1)
                {     // Cash payment
                    payments[i] = new Payment(amount);
                }
                else if (choice == 2)
                {
                    // Check Payment
                    Console.Write("Enter check number: ");
                    int checkNumber = int.Parse(Console.ReadLine());
                    payments[i] = new CheckPayment(amount, checkNumber);
                }
```

Lab 8 Solution (Continued)

```
else if (choice == 3)
                {
                    // Credit Card Payment
                    Console.Write("Enter credit card number: ");
                    string creditCardNumber = Console.ReadLine();
                    payments[i] = new CreditCardPayment
                    (amount, creditCardNumber);
                }
            } // end for
            // Display payments
            Console.WriteLine();
            Console.WriteLine("Payment summary");
            foreach (Payment p in payments)
            {
                Console.WriteLine(p);
            }
        }
    } // end of class PaymentApp
}
```

Lab 10 Solution

```
// Lab 10
// Account.cs
using System;
namespace lab10
{
    public class OverdraftException : Exception
    {
        // constructors
        public OverdraftException()    { }
        public OverdraftException(string message) : base(message) { }
        // Just in case we generate another kind of exception
        // but we must throw an OverdraftException
        public OverdraftException(string message, Exception inner)
            : base(message, inner) { }
    }
    public class Account
    {   // attributes
        private string name;
        private double balance;
        // interest rate for accounts
        private static double rate;
        // constructors
        public Account() : this("", 0.0) { }
        public Account(string name, double balance)
        {
            Name = name; Balance = balance;
        }
        // properties
        public String Name
        {
            get { return name; }
            set { name = value; }
        }
        public double Balance
        {
            get { return balance; }
            set { balance = value; }
        }
        public static double Rate
        {
            get { return rate; }
            set { rate = value; }
        }
```

Lab 10 Solution (Continued)

```
// ToString
        public override string ToString()
        {
            return String.Format(
                "Name is {0}, Balance is {1:C}", Name, Balance);
        }
        // Business methods
        public void Deposit(double amount) { Balance += amount; }
        public void Withdraw(double amount)
        {
            if (amount > Balance)
            {
                throw new OverdraftException(
                    "Withdraw declined: insufficient funds");
            } Balance -= amount;
        }
        public void MonthlyInterest()
        {
            Balance += Balance * (Rate / 12);
        }
    } // end of class Account
    public class AccountApp
    {
        public static void Main()
        {
            // Set the interest rate
            Account.Rate = 0.05;
            // Ask user for their name
            Console.Write("Enter your name: ");
            string name = Console.ReadLine();
            // Ask user for their initial balance
            Console.Write("Enter your initial balance: ");
            double balance = double.Parse(Console.ReadLine());
            Account myAccount = new Account(name, balance);
            int choice;
```

Lab 10 Solution (Continued)

```
do
            {
                Console.WriteLine();
                Console.WriteLine("1) Deposit");
                Console.WriteLine("2) Withdraw");
                Console.WriteLine("3) Balance Inquiry");
                Console.WriteLine("4) Compute Monthly Interest");
                Console.WriteLine("5) Exit");
                Console.Write("Enter choice: ");
                choice = int.Parse(Console.ReadLine());
                while (choice < 1 || choice > 5)

 Console.Write("Invalid choice. Please enter valid choice: ");
                    choice = int.Parse(Console.ReadLine());
                }
                if (choice == 1)
                {
                    Console.Write("Enter amount to deposit: ");
                    double amount = double.Parse(Console.ReadLine());
                    myAccount.Deposit(amount);
                    Console.WriteLine(myAccount);
                }
                else if (choice == 2)
                {
                    try
                    {
                        Console.Write("Enter amount to withdraw: ");
                        double amount =
                            double.Parse(Console.ReadLine());
                        myAccount.Withdraw(amount);
                        Console.WriteLine(myAccount);
                    }
                    catch (OverdraftException e)
                    {
                        Console.WriteLine(e.Message);
                        Console.WriteLine(e.StackTrace);
                    }
                }
                else if (choice == 3)
                {
                    Console.WriteLine(myAccount);
                }
                else if (choice == 4)
                {
myAccount.MonthlyInterest();
                    Console.WriteLine(myAccount);
                }
            } while (choice != 5);
        }
    }// end of class AccountApp
}
```

Lab 11 Solution

```
// Lab 11
// Bookstore.cs
using System;
using System.Collections;
namespace lab11
{
    public class Book
    {
        // attributes
        private string isbn;
        private string title;
        private string author;
        private double price;
        // constructors
        public Book() : this("", "", "", 0.0)
        { }
        public Book(string isbn, string title, string author, double price)
        {
            this.isbn = isbn;
            this.title = title;
            this.author = author;
            this.price = price;
        }
        // properties
        public string Isbn
        {
            get { return isbn; }
            set { isbn = value; }
        }
        public string Title
        {
            get { return title; }
            set { title = value; }
        }
        public string Author
        {
            get { return author; }
            set { author = value; }
        }
        public double Price
        {
            get { return price; }
            set { price = value; }
        }
```

Lab 11 Solution (Continued)

```
// ToString
        public override string ToString()
        {
            return String.Format(
                "Isbn = {0}, Title = {1}, Author = {2}, Price = {3:C}"
                , isbn, title, author, price);
        }
    } // end class Book
    public class Bookstore
    {
        public static void Main()
        {
            Hashtable ht = new Hashtable();
            int choice;
            do
            {
                Console.WriteLine();
                Console.WriteLine("1) Add book");
                Console.WriteLine("2) Remove book");
                Console.WriteLine("3) Display all books");
                Console.WriteLine("4) Exit");
                Console.Write("Enter choice: ");
                choice = int.Parse(Console.ReadLine());
                while (choice < 1 || choice > 4)
                {
                   Console.Write(
                   "Invalid choice. Please enter a value between 1 and 4: ");
                   choice = int.Parse(Console.ReadLine());
                }
```

Lab 11 Solution (Continued)

```
switch (choice)
                {
                    case 1:
                        Console.Write("Enter isdn: ");
                        string isdn = Console.ReadLine();
                        Console.Write("Enter title: ");
                        string title = Console.ReadLine();
                        Console.Write("Enter author: ");
                        string author = Console.ReadLine();
                        Console.Write("Enter price: ");
                        double price = double.Parse(Console.ReadLine());
                        ht.Add(isdn, new Book(isdn, title, author, price));
                        break;
                    case 2:
                        Console.Write("Enter isdn: ");
                        string key = Console.ReadLine();
                        Book b = (Book)ht[key];
                        if (b != null)
                        {
                            ht.Remove(key);
                        }
                        else
                        {
                            Console.WriteLine("Book not found.");
                        } break;
                    case 3:
                        IEnumerator e = ht.Values.GetEnumerator();
                        while (e.MoveNext())
                        {
                            Console.WriteLine(e.Current);
                        } break;
                } // end switch
            }
            while (choice != 4);
        } // end Main
    } // end class Bookstore
} // end namespace
```