Chapter 2

C# Overview for the Sophisticated Programmer

C# Overview for the Sophisticated Programmer

Objectives

After completing this unit you will be able to:

- **Compile and run C# programs in your local development environment.**
- **Describe the basic structure of C# programs.**
- **Describe how related C# classes can be grouped into namespaces.**
- **Describe objects and classes in C#.**
- **Perform input and output in C#.**
- **Outline the principle control structures and operators in C#.**
- **Outline the principle data types in C#.**
- **Describe the difference between value and reference types, and explain how C# achieves a unified type system through "boxing" and "unboxing."**
- **Describe parameter passing in C#.**
- **Use structures, strings and arrays.**
- **Perform formatting in C#.**
- **Use exceptions in C#.**

Hello, World

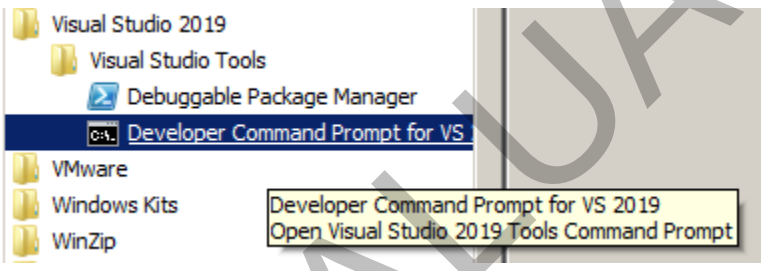
- **Whenever learning a new programming language, a good first step is to write and run a simple program that will display a single line of text.**
  - Such a program demonstrates the basic structure of the language, including output.
  - You must learn the pragmatics of compiling and running the program.
- **Here is "Hello, World" in C#:**
  - See **Hello\Hello.cs** in the **Chap02** directory.

```
// Hello.cs

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

Compiling, Running (Command Line)

- **The Visual Studio 2019 IDE (integrated development environment) was introduced in Chapter 1, and we will use it throughout the course.**
  - See Appendix A for more details.
  - To open an existing project or solution, use the menu File | Open | Project/Solution. You can then navigate to a **.csproj** or **.sln** file.
- **If you are using the .NET SDK, you may do the following:**
  - Open up Developer Command Prompt for VS 2019.



  - Compile the program via the command line:

```
csc Hello.cs
```

  - An executable file **Hello.exe** will be generated. To execute your program, type at the command line:

```
Hello
```

  - The program will now execute, and you should see the greeting displayed. That's all there is to it!

```
Hello, World
```

Program Structure

```
// Hello.cs

class Hello
{
    ...
}
```

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

Program Structure (Cont'd)

```
// Hello.cs

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

- **There is a distinguished class which has a method whose name must be *Main*.**
  - The method should be **public** and **static**.
  - An **int** exit code can be returned to the operating system. Use **void** if you do not return an exit code.

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

  - Command line arguments are passed as an array of strings.
  - The argument list can be empty:

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

  - The runtime will call this **Main** method—it is the entry point for the program.
  - All of the code for the **Main** method will be between the curly braces.
  - Note that in C#, it is not necessary for the file name to be the same as the name of the class containing the **Main** method.

Program Structure (Cont'd)

```
// Hello.cs

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

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

Namespaces

- **Much standard functionality in C# is provided through many classes in the .NET Framework.**
- **Related classes are grouped into *namespaces*.**
- **The fully qualified name of a class is specified by the namespace, followed by a dot, followed by class name.**

```
System.Console
```

- **A *using* statement allows a class to be referred to by its class name alone.**
  - See **Hello2\Hello2.cs**.

```
// Hello2.cs

using System;

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

Variables

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

```
int fahr = 86;        // reserves space and assigns
                      // an intial value
int celsius;          // reserves space but does
                      // not initialize
```

  - If an initial value is not specified, C# initializes the variable to a default value, such as 0.

Input in C#

- **A useful program in C# will typically perform some input.**
- **An easy, uniform way to obtain input for various data types is to read the data in as a string and then convert it to the desired data type.**
  - Use **ReadLine()** method of **System.Console** class to read in a string.
  - Use **ToXxxx()** methods of **System.Convert** class to convert the data.

```
Console.WriteLine("How many temperatures? ");
string buf = Console.ReadLine();
int numTemp = Convert.ToInt32(buf);
```

- **Because this pattern of prompting for input and reading it is common in our console programs, we provide a simple *InputWrapper* class to shorten our code.**

More about Classes

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

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

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

InputWrapper Class

- **The *InputWrapper* class "wraps" interactive input for several basic data types.**
  - The supported data types are **int**, **double**, **decimal,** and **string**.
  - Methods **getInt**, **getDouble**, **getDecimal,** and **getString** are provided.
  - A prompt string is passed as an input parameter.
  - See the file **InputWrapper.cs** in directory **InputWrapper,** which implements the class, and **TestInputWrapper.cs,** which tests the class.
- **You do not need to be familiar with the implementation of *InputWrapper* in order to use it.**
  - That is the beauty of "encapsulation"— complex functionality can be hidden by an easy-to-use interface.

Sample Program

- **This program will convert from Fahrenheit to Celsius.**
  - See **Convert:**

```
// Convert.cs
//
using System;

class ConvertTemp
{
  public static void Main(string[] args)
  {
    // Input is done directly
    Console.Write("Temperature in Fahrenheit: ");
    string buf = Console.ReadLine();
    int fahr = Convert.ToInt32(buf);

    int celsius = (fahr - 32) * 5 / 9;
    Console.WriteLine("fahrenheit = {0}", fahr);
    Console.WriteLine("celsius = {0}", celsius);

    // Use the InputWrapper class
    InputWrapper iw = new InputWrapper();
    fahr = iw.getInt(
      "Temperature in Fahrenheit: ");

    celsius = (fahr - 32) * 5 / 9;
    Console.WriteLine("fahrenheit = {0}", fahr);
    Console.WriteLine("celsius = {0}", celsius);
  }
}
```

Input Wrapper Implementation

```
// InputWrapper.cs
//
// Class to wrap simple stream input
// Datatypes supported:
//      int
//      double
//      decimal
//      string
using System;

class InputWrapper
{
  public int getInt(string prompt)
  {
    Console.Write(prompt);
    string buf = Console.ReadLine();
    return Convert.ToInt32(buf);
  }
  public double getDouble(string prompt)
  {
    Console.Write(prompt);
    string buf = Console.ReadLine();
    return Convert.ToDouble(buf);
  }
  public decimal getDecimal(string prompt)
  {
    Console.Write(prompt);
    string buf = Console.ReadLine();
    return Convert.ToDecimal(buf);
  }
  public string getString(string prompt)
  {
    Console.Write(prompt);
    string buf = Console.ReadLine();
    return buf;
  }
}
```

Compiling Multiple Files

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

```
csc /out:Convert.exe *.cs
```

  - This will compile all of the files in the current directory.
  - The **/out** option specifies the name of the .EXE file.

```
Directory of C:\OIC\CsEss\Chap02\Convert

02/18/2019  01:56 PM  <DIR>          .
02/18/2019  01:56 PM  <DIR>          ..
02/18/2019  01:40 PM               144 app.config
02/18/2019  01:56 PM  <DIR>          bin
07/19/2001  08:33 PM               909 Convert.cs
02/18/2019  01:40 PM             5,407 Convert.csproj
02/18/2019  01:56 PM             4,608 Convert.exe
11/13/2009  01:10 PM               898 Convert.sln
11/13/2009  01:38 PM            11,264 Convert.suo
05/17/2001  10:23 AM               747 InputWrapper.cs
02/18/2019  01:56 PM  <DIR>          obj
               7 File(s)         23,977 bytes
               4 Dir(s)  36,389,105,664 bytes free
```

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

```
csc /out:Convert.exe *.cs /main:ConvertTemp
```

Control Structures

- **C# has the familiar control structures of the C family of languages:**
  - if
  - while
  - do
  - for
  - switch
  - break
  - continue
  - return
  - goto
- **Except for *switch*, which is less error-prone in C#, these controls all have standard semantics.**
- **There is also a *foreach* statement, which we will discuss later in connection with arrays and collections.**
- **The *throw* statement is used with exceptions.**
- **The *lock* statement can be used to enforce synchronization in multi-threading situations.**

switch

- **In C#, after a particular case statement is executed, control does not automatically continue to the next statement.**
  - You must explicitly specify the next statement, typically with a **break** or **goto** *label*.

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

- **You may also switch on a *string* data type.**

C# Operators

- **The C# operators are similar to those in C and C++, with similar precedence and associativity rules.**
- **There are three kinds of operators.**
  - **Unary** operators take one operand and use prefix notation (e.g. **– a**) or postfix notation (e.g. **a++**).
  - **Binary** operators take two operands and use infix notation (e.g. **a + b**).
  - The one **ternary** operator **? :** takes three operands and uses infix notation (e.g. **expr ? x : y**).
- **Operators are applied in the precedence order shown on the next page.**
- **For operators of the same precedence, order is determined by associativity.**
  - The assignment operator is right-associative (operations are performed from right to left).
  - All other binary operators are left-associative (operations are performed from left to right).
- **Precedence and associativity can be controlled by parentheses. The parentheses indicate which operation is performed first, shown as the primary operator (x) in the precedence table.**
- **C# has operators *checked* and *unchecked*, which will be discussed later.**

Precedence Table

- **Precedence goes from the top (highest) to bottom (lowest).**

Category	Operators
Primary	(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked
Unary	+ - ! ~ ++x --x (T)x
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational	< > <= >= is
Equality	== !=
Logical AND	&
Logical XOR	^
Logical OR	|
Conditional	&&
Conditional OR	||
Conditional	?:
Assignment	= *= /= += -= <<= >>= &= ^= |=

Types in C#

- **There are three kinds of types:**
  - Value types
  - Reference types
  - Pointer types
- **Value types directly contain their data.**
  - Each variable of a value type has its own copy of the data.
  - Value types are typically allocated on the stack and get automatically destroyed when the variable goes out of scope.
- **Reference types do not contain data directly, but only refer to data.**
  - Variables of reference types store *references* to data, called *objects*.
  - Two different variables can reference the same data.
  - Reference types are typically allocated on the heap and eventually get destroyed through a process known as *garbage collection*[1].
- **Pointer types are only used in *unsafe* code.**
  - Appendix C discusses pointers and unsafe code.

[1] For a discussion of garbage collection see Chapter 6 of Object Innovations' course .NET Framework Using C#.

Simple Types

- **The simple data types are general-purpose, value data types, including numeric, character, and boolean.**
  - The **sbyte** data type is an 8-bit signed integer.
  - The **byte** data type is an 8-bit unsigned integer.
  - The **short** data type is a 16-bit signed integer.
  - The **ushort** 16-bit unsigned integer.
  - The **int** data type is a 32-bit signed integer.
  - The **uint** 32-bit unsigned integer.
  - The **long** data type is a 64-bit signed integer.
  - The **ulong** 64-bit unsigned integer.
  - The **char** data type is a Unicode character (16 bits).
  - The **float** data type is a single-precision floating point.
  - The **double** data type is a double-precision floating point.
  - The **bool** data type is a Boolean (**true** or **false**).
  - The **decimal** data type is a decimal type with 28 significant digits (typically used for financial purposes).

Types in System Namespace

- **There is an exact correspondence between the simple C# types and types in the *System* namespace.**
  - C# reserved words are simply aliases for the corresponding type in the **System** namespace.

C# Reserved Word	Type in System Namespace
sybte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

Integer Data Types

- **C# defines the following 9 integral data types:**
  - The **sbyte** type is a signed 8-bit integer with the range of –128 to 127, inclusive.
  - The **short** type is a signed 16-bit integer with the range of –32768 to 32767, inclusive.
  - The **int** type is a signed 32-bit integer with the range of –2147483648 to 2147483647, inclusive.
  - The **long** type is a signed 64-bit integer with the range of –9223372036854775808 to 9223372036854775807, inclusive.
  - The **byte** type is an unsigned 8-bit integer with the range of 0 to 255, inclusive.
  - The **ushort** type is an unsigned 16-bit integer with the range of 0 to 65535, inclusive.
  - The **uint** type is an unsigned 32-bit integer with the range of 0 to 4294967295, inclusive.
  - The **ulong** type is an unsigned 64-bit integer with the range of 0 to 18446744073709551615, inclusive.
  - The **char** type is an unsigned 16-bit integer with the range of 0 to 65535, inclusive. This set of values represents the Unicode character set.

Floating Point Data Types

- **C# supports the following pre-defined floating point data types.**
  - The **float** data type is a single-precision floating point.
  - The **double** data type is a double-precision floating point.
- **The float data type is represented in the IEEE 754 32-bit single-precision floating point format.**
- **The double data type is represented in the IEEE 754 64-bit double-precision floating point format.**
- **IEEE 754 define the following special floating point values:**
  - **Positive zero** results from dividing 0.0 by a non-zero positive value.
  - **Negative zero** results from dividing 0.0 by a non-zero negative value.
  - **Positive infinity** results from dividing a non-zero positive value by 0.0.
  - **Negative infinity** results from dividing a non-zero negative value by 0.0.
  - **Not-a-Number (**also known as **NaN**) results from dividing 0.0 by 0.0.

Implicit Conversions

- ***Implicit conversions*** **are provided by the compiler automatically where they are required.**
- **Implicit conversions are guaranteed to be safe, in that no loss of information can occur.**
- **For example, the conversions from *int* to *long*, or from *float* to *double*, are implicit conversions, which are inherently safe.**
  - This is because all values that can be represented by an **int** can be precisely represented by a **long**, and all values that can be represented by a **float** can be precisely represented by a **double**.

Explicit Conversions

- ***Explicit conversions*** **are performed only where the programmer uses a cast expression explicitly.**
- **Explicit conversions are risky, in that loss of information can easily occur.**
- **Special care may need to be taken when explicitly casting an expression.**
  - For example, the conversions from **long** to **int,** or from **double** to **float,** are explicit conversions, which are inherently risky.
  - This is because not all values that can be represented by a **long** can be precisely represented by an **int**, and not all values that can be represented by a **double** can be precisely represented by a **float**.
- **An explicit conversion can also be used to force the compiler to perform the desired type of arithmetic operation (e.g. floating point division).**

```
// Cast one of the integers to double and use
// a double variable for celsius

double dblCel = (fahr - (double) 32) * 5 / 9;
```

- **If an expression attempts to use an unsafe conversion, and the programmer has not provided an explicit cast to perform the conversion, then a compiler error will be generated.**

Boolean Data Type

- **The *bool* data type represents a Boolean value.**
  - Boolean values are also known as logical values and may only be set to the values **true** or **false**.
- **No predefined conversions exist between *bool* and other types.**
- **In C and C++, there are implicit conversions.**
  - An integer value of 0 or pointer value of **null** converts to **false**.
  - A non-zero or non-null value converts to **true**.
- **In C#, you have to explicitly use relational operators.**

```
if (numTemp == 0)
    ...

if (objRef != null)
    ...
```

struct

- **A *struct* is a value type which can group inhomogeneous types together.**
  - It can also have constructors and methods, which we will look at later.

```
public struct Hotel
{
    public string city;
    public string name;
    public int rooms;
    public decimal cost;
}
```

- **A *struct* object is created using the *new* operator.**

```
Hotel hotel = new Hotel();
```

- **A *struct* object can also be created without *new*, but then the fields will be unassigned, and the object cannot be used until the fields have been initialized.**

```
Hotel hotel;
hotel.name = "Sheraton";
// Now it is OK to use hotel.name field
hotel.city = "Atlanta";
hotel.rooms = 100;
hotel.cost = 50.00m;
// Now it is OK to use hotel object
```

Uninitialized Variables

- **The C# compiler will detect attempts to use uninitialized variables.**
  - A **struct** object cannot be used until its fields have been assigned.
  - A simple variable must be initialized before it can be used.

```
int x;
Console.WriteLine("x = " + x);    // error
```

Enumeration Types

- **Finally, an *enumeration* type is a distinct value type with named constants.**
- **An enumeration type is a distinct type with named constants.**
- **Every enumeration type has an underlying type, which is one of:**
  - **byte**
  - **short**
  - **int**
  - **long**
- **An enumeration type is defined through an *enum* declaration.**

```
public enum BookingStatus : byte
{
    HotelNotFound,        // 0 implicitly
    RoomsNotAvailable,    // 1 implicitly
    Ok = 5                // explicit value
}
```

  - If the type is not specified, **int** is used.
  - By default, the first **enum** member is assigned the value 0, the second member 1, etc.
  - Constant values can be explicitly assigned.

Nullable Types

- **Sometimes it is convenient to allow a special null value for a variable, as well as the range of values allowed by the underlying type.**
  - A good example is in databases, where a null value is typically used to represent missing data.
- **You can declare a variable to be nullable by placing a question mark after the data type.**

```
int? number;
```

  - This is equivalent to using **System.Nullable**.

```
System.Nullable<int> number;
```

- **You can then test whether the variable has this special null value by using the property *HasValue*.**
- **See the example program *Nullable*.**

```
public static void Main()
{
   int? number = null;
   ShowNumber(number);
   number = 37;
   ShowNumber(number);
}
private static void ShowNumber(int? number)
{
   if (number.HasValue)
      Console.WriteLine(number);
   else
      Console.WriteLine("UNDEFINED");
}
```

Reference Types

- **A variable of a reference type does not directly contain its data, but instead provides a *reference* to the data stored elsewhere (on the heap).**
- **In C#, there are the following kinds of reference types:**
  - Class
  - Array
  - Interface
  - Delegate
- **Reference types have a special value, *null*, which indicates the absence of an instance.**

Class Types

- **A class type defines a data structure that has data members, function members, and nested types.**
- **Class types support *inheritance*.**
  - Through inheritance, a derived class can extend or specialize a base class.
  - We will discuss inheritance and other details about classes in the next chapter.

object

- **The *object* class type is the ultimate base type for all types in C#.**
  - Every C# type derives directly or indirectly from **object**.
- **The *object* keyword in C# is an alias for the predefined *System.Object* class.**
- ***System.Object* has methods such as *ToString()*, *Equals()* and *Finalize()*, which we will study later.**

string

- **The *string* class encapsulates a Unicode character string.**
- **The string keyword is an alias for the pre-defined *System.String* class.**
- **The *string* type is a sealed class.**
  - A **sealed** class is one that cannot be used as the base class for any other classes.
- **The *string* class inherits directly from the root *object* class.**
- **String literals are defined using double quotes.**
- **There are useful built-in methods for *string*.**
  - For now, note that the **Equals()** method can be used to test for equality of strings.

```
string a = "hello";
if (a.Equals("hello"))
    Console.WriteLine("equal");
else
    Console.WriteLine("not equal");
```

- **There are also overloaded operators:**

```
if (a == "hello")
    ...
```

Copying Strings

- **Recall that C# has value types and reference types.**
  - A value type contains all of its own data.
  - A reference type refers to data stored somewhere else.
- **As a class, *string* is a reference type.**
- **If a reference variable gets copied to another reference variable, both will refer to the same object.**
- **If the object referenced by the second variable is changed, the first variable will also reflect the new value.**

```
string s1 = "hello";
string s2 = s1;      // s2 also refers to "hello"
```

- **To provide more predictable program behavior, strings in C# are *immutable*.**
  - Once assigned a value, the object a string refers to cannot be changed.
  - What you may think of as changing the value of a string is really giving a new reference.

```
string s = "bat";
s = s + "man";   // a new object is created and
                 // s is assigned to refer to this
                 // new object
```

StringBuilder Class

- **As we have just discussed, instances of the *string* class are immutable.**
  - As a result, when you manipulate instances of **string**, you are frequently obtaining new **string** instances.
  - Depending on your applications, creating all of these instances may be expensive.
  - The .NET library provides a special class, **StringBuilder** (located in the **System.Text** namespace), in which you may directly manipulate the underlying string without creating a new instance.
  - When you are done, you can create a **string** instance out of an instance of **StringBuilder** by using the **ToString()** method.
- **A *StringBuilder* instance has a capacity and a maximum capacity.**
  - These capacities can be specified in a constructor when the instance is created.
  - By default, an empty **StringBuilde**r instance starts out with a capacity of 16.
  - As the stored string expands, the capacity will be increased automatically.

StringBuilderDemo

- **The program *StringBuilderDemo* provides a simple demonstration of using the *StringBuilder* class.**
  - It shows the starting capacity and the capacity after strings are appended. At the end, a **string** is returned.

```
// StringBuilderDemo.cs

using System;
using System.Text;

public class StringBuilderDemo
{
   public static void Main(string[] args)
   {
      StringBuilder build = new StringBuilder();
      Console.WriteLine("capacity = {0}",
         build.Capacity);
      build.Append(
         "This is the first sentence.\n");
      Console.WriteLine("capacity = {0}",
         build.Capacity);
      build.Append(
         "This is the second sentence.\n");
      Console.WriteLine("capacity = {0}",
         build.Capacity);
      build.Append("This is the last sentence.\n");
      Console.WriteLine("capacity = {0}",
         build.Capacity);
      string str = build.ToString();
      Console.Write(str);
   }
}
```

Classes and Structs

- **While in C++ the concept of *class* and *struct* is very close, there is more of a fundamental difference between them in C#.**
  - In C++, a class has default visibility of private and a struct has default visibility of public, and that is the *only* difference.
- **In C#, the key difference between a class and a struct is that a class is a reference type and a struct is a value type.**
- **A class must be instantiated explicitly, using *new*.**
  - The new instance is created on the heap, and memory is managed by the system through a garbage collection process.
- **A struct instance may simply be declared, or you may use *new*.**
  - For a struct, the new instance is created on the stack, and the instance will be deallocated when it goes out of scope.
- **There are different semantics for assignment, whether done explicitly or via call-by-value mechanism in a method call.**
  - For a class, you will get a second object reference and both object references refer to the same data.
  - For a struct, you will get a completely independent copy of the data in the struct.

Static and Instance Methods

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

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

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

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

Method Parameters

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

No “Freestanding” Functions in C#

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

Classes with All Static Methods

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

```
// SimpleMath.cs

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

Parameter Passing

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

Parameter Terminology

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

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

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

Value Parameters

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

Reference Parameters

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

Reference Parameters Example

- **The program *RefOutMath* illustrates using *ref* parameters.**
  - A single method **Calculate** passes back two values as reference parameters.

```
static void Main(string[] args)
{
// ref keyword is used in front of the arguments
// Variables must be initialized before used as
// ref arguments
   int sum = 0, product = 0;
   Calculate(5, 7, ref sum, ref product);
   Console.WriteLine("sum = {0}", sum);
   Console.WriteLine("product = {0}", product);
   ...

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

Output Parameters

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

Output Parameters Example

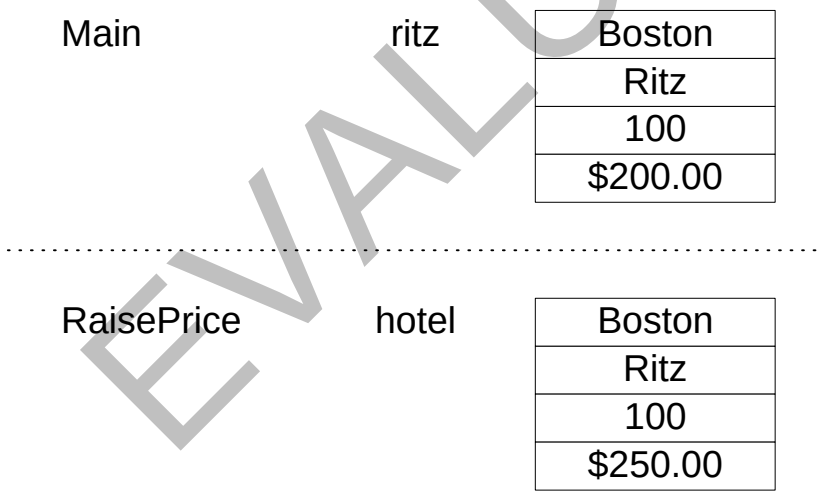
- **The program *RefOutMath* also illustrates using *out* parameters.**
  - A second method **Calculate2** passes back two values as output parameters.

```
static void Main(string[] args)
{
   ...
// out keyword is used in front of the arguments
// Variables need not be initialized before used as
// out arguments
   int sum2, product2;
   Calculate2(15, 7, out sum2, out product2);
   Console.WriteLine("sum = {0}", sum2);
   Console.WriteLine("product = {0}", product2);
...

// You cannot define overloaded methods that differ
// only on ref and out
static void Calculate2(int x, int y, out int sum,
out int prod)
{
   sum = x + y;
   prod = x * y;
}
```

Structure Parameters

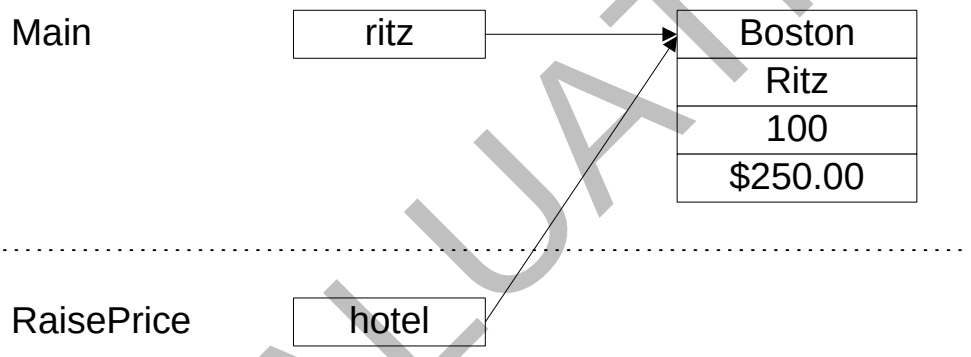
- **A struct is a value type, so that if you pass a struct as a value parameter, the struct instance in the called method will be an independent copy of the struct in the calling method.**
- **The program *HotelStruct* illustrates passing an instance of a *Hotel* struct by value.**
- **The object *hotel*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



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

Class Parameters

- **A class is a reference type, so that if you pass a class instance as a value parameter, the class instance in the called method will refer to the same object as the reference in the calling method.**
- **The program *HotelClass* illustrates passing an instance of a *Hotel* class by value.**
  - This figure illustrates how the **hotel** reference in the **RaisePrice()** method refers to the same object as the **ritz** reference in **Main()**.



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

Method Overloading

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

Method Overloading (Cont'd)

- **At runtime, the compiler will resolve a given invocation of the method by trying to match up the actual parameters with formal parameters.**
  - A match occurs if the parameters match exactly or if they can match through an implicit conversion.
  - For the exact matching rules, consult the C# Language Specification.
- **The program *OverloadDemo* illustrates method overloading.**
  - The method **FindMax()** is overloaded to take either **long** or **string** parameters.
  - The method is invoked three times, for **int**, **long**, and **string** parameters.
  - There is an exact match for the case of **long** and **string**.
  - The call with **int** actual parameters can resolve to the **long** version, because there is an implicit conversion of **int** into **long**.

Variable Length Parameter Lists

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

Arrays

- **An array is a collection of elements with the following characteristics:**
  - All array elements must be of the same type. The element type of an array can be any type, including an array type. An array of arrays is often referred to as a jagged array.
  - An array may have one or more dimensions. For example, a two-dimensional array can be visualized as a table of values. The number of dimensions is known as the array's rank.
  - Array elements are accessed using one or more computed integer values, each of which is known as an index. A one-dimensional array has one index.
  - In C#, an array index starts at 0, as in other C family languages.
  - The elements of an array are created when the array object is created. The elements are automatically destroyed when there are no longer any references to the array object.

One-Dimensional Arrays

- **An array is declared using square brackets [] after the type, not after the variable.**

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

  - Note that the *size* of the array is not part of its type.
  - The variable declared is a *reference* to the array.

- **You create the array elements and establish the size of the array using the *new* operator.**

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

  - The new array elements start out with the appropriate default values for the type (0 for **int**).

- **You may both declare and initialize array elements using curly brackets, as in C/C++.**

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

- **You can indicate you are done with the array elements by assigning the array reference to *null*.**

```
a = null;
```

  - The garbage collector is now free to deallocate the elements.

System.Array

- **Arrays are objects.**
  - **System.Array** is the abstract base class for all array types.
- **Accordingly, you can use the properties and methods of *System.Array* for any array.**

```
Array.Sort(a);        // sorts the array
for (int i = 0; i < a.Length; i++)
    Console.Write("{0} ", a[i]);
Console.WriteLine();
```

- **For a sample array program, see *ArrayDemo*.**

Jagged Arrays

- **You can declare an array of arrays, or a "jagged" array.**
  - Each row can have a different number of elements.

```
int [][] binomial;
```

- **You then create the array of rows, specifying how many rows there are (each row is itself an array):**

```
binomial = new int [rows][];
```

- **Next you create the individual rows:**

```
binomial[i] = new int [i+1];
```

- **Finally, you can assign individual array elements:**

```
binomial[0][0] = 1;
```

- **The example program creates and prints Pascal's triangle.**
  - See **Pascal**.

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

- **Higher-dimensional jagged arrays can be created following the same principles.**

Rectangular Arrays

- **C# also permits you to define rectangular arrays.**
  - All rows have the same number of elements.
- **First you declare the array:**

```
int [,] MultTable;
```

- **Then you create all of the array elements, specifying the number of rows and columns:**

```
MultTable = new int[rows, columns];
```

- **Finally, you can assign individual array elements:**

```
MultTable[i,j] = i * j;
```

- **The *RectangularArray* program creates and prints out a multiplication table.**

```
 0   0   0   0   0
 0   1   2   3   4
 0   2   4   6   8
 0   3   6   9  12
 0   4   8  12  16
```

- **Higher dimensional rectangular arrays can be created following the same principles.**

foreach for Arrays

- **C# provides a *foreach* loop that can be used to iterate through the elements of an array.**
- **The sample code used nested *foreach* loops to print all of the elements of a jagged array on the same line.**
  - See **Pascal**.

```
foreach (int[] row in binomial)
{
    foreach (int x in row)
    {
        Console.Write("{0} ", x);
    }
    Console.WriteLine();
}
```

Boxing and Unboxing

- **One of the strong features of C# is that it has a unified type system.**
- ***Every* type, including the simple built-in types, such as *int*, derive from *System.Object*.**
  - In C#, everything is an object.
- **A language such as Smalltalk also has such a feature, but pays the price of inefficiency for simple types.**
- **Languages such as C++ and Java (before Java 5.0) treat simple built-in types differently than objects, thus obtaining efficiency, but at the loss of a unified type system.**
- **C# enjoys the best of both worlds through a process known as "boxing."**
  - "Boxing" converts a value type, such as an **int** or a **struct,** to an object and does so implicitly.
  - "Unboxing" converts a boxed value type (stored on the heap) back to an unboxed, simple value (stored on the stack). Unboxing is done through a type cast.

```
int x = 5;
object o = x;         // boxing
x = (int) o;          // unboxing
```

- **But there is a performance penalty from boxing and unboxing.**

Implicitly Typed Variables

- **The *var* keyword lets you declare and initialize a variable without explicitly specifying a type.**
  - But the variable still has a type, inferred from the expression on the right-hand side.

```
var num = 55;
// type is Int32

var word = "Hello!";
// type is String
```

  - The C# **var** is *not* a "variant" data type, such as **var** in JavaScript.

- **The *var* keyword can also be used to declare and initialize an array.**

```
var primes = new[] { 2, 3, 5, 7, 11 };
// type is Int32[]
```

Implicitly Typed Variables – Example

- **See *ImplicitType*.**

```
static void Main(string[] args)
{
   var num = 55;
   ShowObject(num);
   ShowTypeInfo(num);

   var word = "Hello!";
   ShowObject(word);
   ShowTypeInfo(word);

   var primes = new[] { 2, 3, 5, 7, 11 };
   ShowArray(primes);
   ShowTypeInfo(primes);

   var words = new [] {"one", "two", "three"};
   ShowArray(words);
   ShowTypeInfo(words);
}
```

- **Here is the output:**

```
55
Type = Int32
Base class = System.ValueType
Hello!
Type = String
Base class = System.Object
2 3 5 7 11
Type = Int32[]
Base class = System.Array
one two three
Type = String[]
Base class = System.Array
```

Output in C#

- **Simple output (e.g. for debugging) for various data types can be done using *Console.WriteLine*() method applied to a *string*.**
  - The **ToString()** method of **System.Object** will provide a string representation for any data type.
  - For custom data types, you should override **ToString()**.
  - You can use the + concatenation operator for strings to build up an output string (a technique that can also be applied in other contexts, such as building a SQL query string).

```
int x = 24;
int y = 26;
Console.WriteLine("Product of " + x + " and "
    + y + " is " + x*y);
```

- **Alternatively, you can use {0}, {1}, etc. as placeholders.**

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

Output:

```
Product of 24 and 26 is 624
```

Formatting

- **C# has extensive formatting capabilities, which you can control through the placeholders.**
  - Simplest: {n}, where n is 0, 1, 2, ...
  - Control width: {n,w}, where w is width (positive for right-justified and negative for left-justified)
  - Format string: {n:S}, where S is a format string
  - Width and format string: {n,w:S}
- **A format string consists of a format character followed (optionally) by a precision specifier.**

Format Character	Meaning
C	Currency (locale specific)
D	Decimal integer
E	Exponential (scientific)
F	Fixed-point
G	General (E or F)
N	Number with embedded commas
X	Hexadecimal

Formatting Example

```
double pi = Math.PI;
decimal cost = 70.45m;
Console.WriteLine("{0,30}", pi);
                                  // width 30
Console.WriteLine("{0,-30}", pi);
                                  // left justified
Console.WriteLine("{0,30:F}", pi);
                                  // fixed point
Console.WriteLine("{0,30:F4}", pi);
                                  // precision 4
Console.WriteLine("{0,30:C}", cost);
                                  // currency
```

Output:

```
            3.14159265358979
3.14159265358979
                          3.14
                        3.1416
                       $70.45
```

– See **FormatDemo**

Exceptions

- **C# provides an exception mechanism similar in concept to exceptions in C++ and Java.**
- **Exceptions are implemented 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 either **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 program, an exception is raised through a **throw** statement.

Exception Example

- **See *ExceptionDemo\Step1*.**

```
using System;

public class ExceptionDemo
{
...
  public static int Main(string[] args)
  {
    int prod;
    long lprod;
    try
    {
        ...
        prod = CheckedMultiply(56666, 57777L);
        Console.WriteLine("product = {0}", prod);
    }
    catch (OverflowException e)
    {
        Console.WriteLine(
            "Overflow Exception: {0}", e.Message);
        Console.WriteLine(
            "Overflow Exception: {0}", e);
    }
    catch (Exception e)
    {
        Console.WriteLine(
            "Exception: {0}", e.Message);
        Console.WriteLine("Exception: {0}", e);
    }
    Console.WriteLine("count = {0}", count);
    return 0;
  }
}
```

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.
- **The following method can throw two different kinds of exceptions:**

```
private static int CheckedMultiply(
                        object a, object b)
{
    int product = checked((int) a * (int) b);
    count++;
    return product;
}
```

  - The type casts can fail, resulting in **InvalidCastException**.
  - The multiplication can overflow, resulting in **OverflowException**.

Throwing New Exceptions

- **In general, it is wise to handle exceptions, at least at some level, near their source.**
  - You have the most information available.
  - See **ExceptionDemo\Step2**.
- **A common pattern is to create a new exception object that captures more detailed information and throw this on to the calling program.**

```
private static int CheckedMultiply(
                                object a, object b)
{
    int first, second;
    try
    {
        first = (int) a;
    }
    catch (InvalidCastException e)
    {
        count++;
        throw new Exception(
           "First operand is not an int", e);
    }
    try
    {
        second = (int) b;
    }
    catch (InvalidCastException e)
    {
        count++;
        throw new Exception(
           "Second operand is not an int", e);
    }
...
```

finally

```
...
    try
    {
        int product = checked(first * second);
        return product;
    }
    catch (OverflowException e)
    {
        throw new Exception(
            "Integer overflow", e);
    }
    finally
    {
        count++;
    }
}
```

- **A *finally* block is always executed when control leaves a *try* block.**
  - In the example above, the counter is always incremented, whether or not an exception occurs.

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.

Lab 2

Implementing a Customers Class

In this lab, you will begin the Acme Travel Agency case study by implementing a simple Customers class in C#. You are provided with starter code that defines a class for an individual customer and a test program. You are to implement a class that can be used by Acme to keep track of customers who register for its services. Customers supply their first and last name, and email address. The system assigns a customer id. The following features are supported:

- Register a customer, returning a customer id
- Unregister a customer
- Obtain customer information, either for a single customer or for all customers (pass the customer id, and for customer id of –1 return all customers)
- Change customer's email address

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

Suggested time: 60 minutes

Summary

- **Every C# application has a class with a method *Main*, which is the entry point into the application.**
- **The *System* class includes methods for doing input and output, such as *ReadLine()* and *WriteLine()*.**
- **The .NET Framework has a large class library that is partitioned into namespaces.**
- **C# has control structures and operators similar to those in C and C++.**
- **C# has value, reference, and pointer data types.**
- **Through boxing and unboxing, C# achieves a unified type system, with all types acting as if they are derived from *object.***
- **Built-in numeric types, *bool*, and *struct* are value types.**
- **Examples of reference types are *object*, *string*, and arrays.**
- **C# has a flexible parameter passing mechanism that can be controlled through *ref* and *out* keywords.**
- **C# has extensive formatting capabilities, which you can control through the placeholders.**
- **Exceptions in C# are implemented by the Common Language Runtime.**

Lab 2

Implementing a Customers Class

Introduction

In this lab, you will begin the Acme Travel Agency case study by implementing a simple Customers class in C#. You are provided with starter code that defines a class for an individual customer and a test program. You are to implement a class that can be used by Acme to keep track of customers who register for its services. Customers supply their first and last name, and email address. The system assigns a customer id. The following features are supported:

- Register a customer, returning a customer id
- Unregister a customer
- Obtain customer information, either for a single customer or for all customers (pass the customer id, and for customer id of –1 return all customers)
- Change customer's email address

Suggested Time: 60 minutes

Root Directory: **OIC\CsEss**

Directories:	**Labs\Lab2\Acme**	(Do your work here)
	CaseStudy\Acme\Step0	(Backup of starter files)
	CaseStudy\Acme\Step1	(Answer)

Files: **Customer.cs**
Test.cs

Instructions

1. Build the starter program. There is a complete implementation of a **Customer** class and a stub implementation of a **Customers** class. There is also a test program. Examine the starter code and run the program. Notice that the test program handles exceptions. For example, the stub **GetCustomer** function returns a **null**, which is checked for in the test program. Also, if you enter non-numeric data when prompted for an id in the test program, an exception will be thrown.

2. Add to the **Customers** class declarations of the following private members: an array **customers** of type **Customer[]** and a variable **nextc** of type **int**. We will use **nextc** as the index of the next element to be added to the array, and it should be initialized to 0.

3. Add code to the **Customers()** constructor that will instantiate the customers array to have 10 elements and register some sample customers.

4. Add code in **RegisterCustomer** to instantiate a new **Customer** with the specified fields, store this customer in the array, increment **nextc**, and return the id of this new customer. (Note that an id is automatically generated by the constructor of **Customer**.)

5. Replace the stub code in **GetCustomer** by code that will assign **count** to be **nextc** and return **customers**. (Temporarily, we are trying to always return the entire array.) Build and test. No customers are being shown as returned. Why?

6. The parameter **count** is passed by value, and so its new value is not passed to the calling program. To fix this, we need to make it either a **ref** or an **out** parameter. Since it only does output, we make it an **out** parameter. Build. We get compiler errors. Why?

7. We also need to use the **out** modifier in the calling program **Test.cs**. Change this in the two places where **GetCustomer** is called. Build and run. Now you should see your sample data returned in response to the "customers" command. Also, the "register" command should be working, so that you can register additional customers.

8. Now we want to provide the full functionality of **GetCustomer**. If id of –1 is passed, the entire array is passed back. Otherwise, an array of 1 element is created, having the customer information for the id that is provided. To implement this feature, first provide code for the helper method **FindId**. This method does a linear search for the given id. If not found, it returns –1. Otherwise it returns the index at which the id was found.

9. Now finish the implementation of **GetCustomer**. Build and test. Now you should be able to query for a single customer by id, as well as obtain the complete list of customers.

10. Implement **UnregisterCustomer**. If the customer is not found, throw an exception. Otherwise, delete the customer from the array. Move the elements after the deleted element up in the array, to fill the deleted item. Build and test.

11. Finally, implement **ChangeEmailAddress**. Build and test. Your miniature customer management system should now be completely working!