SWE344

Internet Protocols and Client-Server Programming

Module 2c: C# Programming Essentials

Dr. El-Sayed El-Alfy alfy@kfupm.edu.sa

Mr. Bashir M. Ghandi bmghandi@ccse.kfupm.edu.sa

Computer Science Department

King Fahd University of Petroleum and Minerals

Objectives

- Learn more about how C# programs are organized
- Learn how to declare Methods and Classes
- Learn how Inheritance and Polymorphism are achieved
- Learn how to declare and implement Interfaces
- Learn about structures (struct) and Enumerators (enum)
- Learn how to raise and handle exceptions
- Learn how to do basic file IO using Streams.

Agenda

- Interfaces
- Structures (struct) and Enumerators (enum)
- Exceptions
- Basic File IO using Streams

Interfaces

- Like Java, Interfaces are used to minimize the effect of lack of multiple inheritance.
- Interfaces contain *only* method specification without implementation. The methods are implicitly public and abstract – declaring them as such is an error.
- Unlike Java, interfaces cannot have even constant fields.
- A class can implement multiple interfaces. However, there is no “implements” keyword. Instead, a colon is used for implements.

Example1

```
1.   using System;
2.   public interface MyComparable {
3.       int CompareTo(Object obj);
4.   }
5.   public abstract class Shape : MyComparable {
6.       public String name() {
7.               return GetType().Name;
8.       }
9.       public abstract double Area();
10.      public abstract double Perimeter();
11.      public override String ToString() {
12.              return "ShapeType:"+name() + ":" + Perimeter() + ":" + Area();
13.      }
14.      public int CompareTo(Object obj) {
15.              Shape shape = (Shape) obj;
16.              if (Area()< shape.Area())
17.                      return -1;
18.              else if (Area() > shape.Area())
19.                      return 1;
20.              else
21.                      return 0;
22.      }
23.  }
```

Example2

```
1.   using System;
2.   namespace Shapes {
3.        public abstract class Shape : IComparable {
4.          public String name() {
5.               return GetType().Name;
6.          }
7.          public abstract double Area();
8.          public abstract double Perimeter();
9.          public override String ToString() {
10.              return "ShapeType:"+name() + ":" + Perimeter() + ":" + Area();
11.         }
12.         public int CompareTo(Object obj) {
13.              Shape shape = (Shape) obj;
14.              if (Area()< shape.Area())
15.                       return -1;
16.              else if (Area() > shape.Area())
17.                       return 1;
18.              else
19.                       return 0;
20.         }
21.       }
```

Example2 …

```
22.       public class Rectangle : Shape {
23.          private double length;
24.          private double width;
25.
26.          public double Length {
27.              get {return length;}
28.              set {length = value;}
29.          }
30.          public double Width {
31.              get {return width;}
32.              set {width = value;}
33.          }
34.          public Rectangle(double length, double width) {
35.              this.length = length;
36.              this.width = width;
37.          }
38.          public override double Area() {
39.           return length*width;
40.          }
41.          public override double Perimeter() {
42.              return 2*length + 2*width;
43.          }
44.       }
```

Example2 …

```
45.       public class Square : Rectangle {
46.          public Square(double length) : base(length, length) {
47.          }
48.       }
49.       public class Circle : Shape {
50.          private double radius;
51.          public double Radius {
52.               get {return radius;}
53.               set {radius = value;}
54.          }
55.          public Circle(double r) {
56.               radius = r;
57.          }
58.          public override double Area() {
59.               return Math.PI * (radius * radius);
60.          }
61.          public override double Perimeter() {
62.               return 2.0 * Math.PI * radius;
63.          }
64.       }
```

Example2 …

```
65.       public class TestShapes {
66.          public static void Main(String[] args) {
67.               Shape[] shape = new Shape[3];
68.               shape[0] = new Rectangle(20, 10);
69.               shape[1] = new Square(10);
70.               shape[2] = new Circle(7);
71.               for (int i=0; i<shape.Length; i++)
72.                        Console.WriteLine(shape[i]);
73.               foreach (Shape s in shape) {
74.                  if (s is Circle) {//using is and as operators
75.                        Circle c = s as Circle;
76.                        Console.WriteLine("The radius is: "+c.Radius);
77.                  }
78.               }
79.               Array.Sort(shape); //sorting the shapes
80.               Console.WriteLine("sorting");
81.               for (int i=0; i<shape.Length; i++)
82.                        Console.WriteLine(shape[i]);
83.          }
84.       }
85.  }
```

```
ShapeType:Rectangle:60:200
ShapeType:Square:40:100
ShapeType:Circle:43.9822971502571:153.9380400259
The radius is: 7
sorting
ShapeType:Square:40:100
ShapeType:Circle:43.9822971502571:153.9380400259
ShapeType:Rectangle:60:200
Press any key to continue . . .
```

Structures

- struct is a lightweight version similar to a class in its declaration and in terms of the members it can have.
- Like a class, struct members can be constructors, constants, fields, methods, properties, indexers, operators, and nested types.
- Structs are treated as value types not reference types hence
  - they are stored in the stack
  - they don't incur the overhead associated with reference objects except when boxed
- Simple example

```
struct Color{
   public int Red;
   public int Green;
   public int Blue;
}
```

Structures …

- Because a struct is a value type, it is allocated memory once it's declared (without using new keyboard)

```
Color rgb;
rgb.Red = 0;
rgb.Green = 0;
rgb.Blue = 0;
```

- A struct can also be initialized using new, e.g.

```
Color rgbColor = new Color();
Console.WriteLine(rgbColor.Red);
```

implicit default constructor that initializes the fields of a struct to their default values.

Structures …

- You can define non-default constructors as well as methods

```
1.   struct Color
2.   {
3.       public int Red;
4.       public int Green;
5.       public int Blue;
6.       public Color(int red, int green, int blue)
7.       {
8.           Red = red;
9.           Green = green;
10.          Blue = blue;
11.      }
12.      public override String ToString()
13.      {
14.          return "(Red="+ Red + ", Green=" + Green + ", Blue=" + Blue+")";
15.      }
16.  }
```

Structures …

- structs have some limitations
  - No support of inheritance (can not be derived from or used to derive other classes or structs)
    - Except System.Value from which all structs are derived
    - But a struct can implement an interface
  - Cannot define a default constructor (it is always defined automatically) but can explicitly define non-default constructors
  - Cannot define a destructor
  - No copy constructor (you can directly use =)
- As a general rule, you should use structs only when:
  - The data being contained is very small,
    - e.g., structs that hold Point values (x and y), RGB Color values.
  - The struct will contain few or even no methods to access or modify the contained data.

Enumerators

- An enumeration (enum) is a special form of value type, which is used to assign symbolic names to a restricted set of values of an underlying integral type (int, uint, byte, sbyte, short, etc – except char).
- An enumeration type has a name, an underlying type, and a set of fields.
  - fields are static literals, each of which represents a constant.
- Example: declare an enum representing week days.

```
1.   public enum WeekDay {
2.       Sunday,
3.       Monday,
4.       Tuesday,
5.       Wednesday,
6.       Thursday,
7.       Friday,
8.       Saturday
9.   }
```

default settings:
the value type is int;
the first literal specified is set to 0 and this value is then incremented for each subsequent literal.

Enumerators …

- To access the value a specific field, use the dot operator
  - E.g., WeekDay.Sunday is the integer 0 and WeekDay.Saturday is the integer 6.
- You can change the type and the value assigned to the first literal, e.g.

```
1.   public enum WeekDay : byte {
2.       Sunday = 1,
3.       Monday,
4.       Tuesday,
5.       Wednesday,
6.       Thursday,
7.       Friday,
8.       Saturday
9.   }
```

Enumerators …

- The advantage of using enum is that they make a program more readable and less error prone than using the underlying values directly.
- For example, compare the following two methods:

```
1.   public static bool IsWeekEnd(WeekDay day) {
2.        return day == WeekDay.Thursday || day == WeekDay.Friday;
3.   }
```

```
1.   public static bool IsWeekEnd(int day) {
2.        return day == 5 || day == 6;
3.   }
```

Enumerators …

- Clearly, the first method (using enum) is more readable.
- Also since the second method takes an int as argument, it is possible to call it with any int value – which could lead to errors -- whereas, the first method can only take one of its specified values.
- System.Enum class provides some methods that can be used to manipulate enum types.
- Example – printing the literals and the values of the WeekDay enumerator.

```
1.   String[] names = Enum.GetNames(typeof(WeekDay));
2.   foreach(string s in names)
3.        Console.WriteLine(s);
4.   byte[] values = (byte[]) Enum.GetValues(typeof(WeekDay));
5.   foreach(byte i in values)
6.        Console.WriteLine(i);
```

Exception Handling

- An exception is an object that encapsulates information about an unusual program occurrence.
- All exceptions in C# are run-time exceptions derived from the System.Exception class.
- An exception is different than a bug and error
  - A *bug* is a programmer mistake that should be fixed before the code is shipped
  - An *error* is caused by user action, e.g. the user might enter a number where a letter is expected
- Bugs and errors can lead to exceptions
- Even if you remove all bugs and anticipate all user errors, you will still run into predictable but unpreventable problems,
  - such as running out of memory or attempting to open a file that no longer exists.
- You can't prevent exceptions, but you can handle them so that they don't bring down your program.

Exception Handling …

- Similar to Java, exceptions are handled using try statement.
- There are three forms of try statement
  - A try block followed by one or more catch blocks.
  - A try block followed by a finally block.
  - A try block followed by one or more catch blocks followed by a finally block.
- Notes
  - the code that may result in exception is placed in the try block.
  - the catch block is used to handle the exception if it occurs.
  - the optional finally block is used to place a code that must be executed whether an exception is raised or not.
- Note also that there are many options for the catch block:
  - If there is no need to refer to the exception instance, then there is no need to declare a variable to receive it.
  - the whole catch expression can be omitted if there is no need to refer to the resulting exception. This will catch all exceptions even those that are not derived from the System.Exception class.
- Finally, we note that c# does not have the throws keyword. If a method does not wish to handle an exception, it just ignores it, and it will automatically be forwarded to a higher method.

Example

```
1.    using System;
2.    public class  TestException {
3.          public static double Divide(double x, double y) {
4.             if (y==0)
5.                   throw new DivideByZeroException("Can't divide by zero");
6.             else
7.                   return x/y;
8.          }
9.          public static void Main() {
10.            try {
11.                  Console.Write("Enter first value: ");
12.                  double x = double.Parse(Console.ReadLine());
13.                  Console.Write("Enter second value: ");
14.                  double y = double.Parse(Console.ReadLine());
15.                  Console.WriteLine(x + "/" + y + " = "+ Divide(x, y));
16.            }
17.            catch (DivideByZeroException e) {
18.                  Console.WriteLine(e.Message);
19.            }
20.            catch (FormatException) {
21.                  Console.WriteLine("Format exception occurs");
22.            }
23.            catch {
24.                  Console.WriteLine("Some Other Exception occurs");
25.            }
26.         }
27.   }
```

uses throw to raise exception when the second number is zero.

I/O Streams

- A stream is a flow of data (sequence of bytes) traveling from a source to a destination
  - The source/destination can be a file, a network connection, or other I/O devices
- I/O operations are designed around streams.
  - Network programming is mainly about Protocols and I/O, so it is important to understand these IO classes.
- The InputSream and OuputSream classes in Java are unified in C# into a single abstract class called Stream.
  - The Stream class defines operations for reading and writing raw, typeless data in the form of bytes
- Once a stream has been opened, it stays open and can be read from or written to until the stream is flushed and closed.
  - Flushing a stream updates the writes made to the stream
  - Closing a stream first flushes the stream, then closes the stream

Example

- An example that creates a text file on disk and uses the abstract Stream type to write data to it

```
1.  using System.IO;
2.  class Test {
3.      static void Main( ) {
4.         Stream s = new FileStream("foo.txt", FileMode.Create);
5.         s.WriteByte(67);
6.         s.WriteByte(35);
7.         s.Close( );
8.      }
9.  }
```

Stream Class Methods

int Read(in byte[] *buffer*, int offset, int count)	Reads *count* bytes from a source and stores the bytes read into the *buffer* array, stating at index *offset*. It returns the number of the actual bytes read or 0
int ReadByte()	Reads one byte from the stream and moves the position by one byte, or returns -1 if at the end of the stream.
void Write(in byte[] *buffer*, int offset, int count)	Writes count bytes from the buffer array, starting at index offset, into a destination
void WriteByte(byte value)	Writes a byte to the current position in the stream and advances the position within the stream by one byte.
void Flush()	Clears all buffers for this stream and causes any buffered data to be written to the underlying device.
void Close()	Closes the current stream and releases any resources associated with the stream.

FileStream Class

- Java provides separate classes for file input and file output (FileInputStream and FileOutputStream)
- In C#, FileStream is used for both input and output.
  - FileStream is a concrete class that extends the Stream class.
  - It allows streams of bytes to be transferred between a source file and a destination file.
- Another concrete class that extends the Stream class is the NetworkStream class.
  - A lot of our network programs will use this class.
- Some Constructors of the FileStream class:

```
public FileStream(string path, FileMode mode)
public FileStream(string path, FileMode mode, FileAccess access)
public FileStream(string path, FileMode mode, FileAccess access, FileShare share)
```

FileStream Class …

- FileMode
  - An enumeration type that is used to indicate how the operating system should open the file.
  - Its values are

Append	Opens the file if it exists and seeks to the end of the file, or creates a new file. FileMode.Append can only be used in conjunction with FileAccess.Write. Any attempt to read fails and throws an ArgumentException.
Create	Specifies that the operating system should create a new file. If the file already exists, it will be overwritten.
CreateNew	Specifies that the operating system should create a new file. If the file already exists, an IOException is thrown.
Open	Specifies that the operating system should open an existing file. A FileNotFoundException is thrown if the file does not exist.
OpenOrCreate	Specifies that the operating system should open a file if it exists; otherwise, a new file should be created.
Truncate	Specifies that the operating system should open an existing file. Once opened, the file should be truncated so that its size is zero bytes. Attempts to read from a file opened with Truncate causes an exception.

FileStream Class …

- FileAccess
  - specify whether the file is being opened for reading, writing or both.
  - The values of the FileAccess enumeration are: Read, Write and ReadWrite.
- FileShare
  - specify how other threads or processes should be allowed access to the same file.
  - The values of the FileShare enumeration are: Inheritable, Read, Write, ReadWrite.
- The methods of FileStream class are essentially the same as those of the Stream class.
- Some useful properties of the FileStream class are: CanRead, CanWrite, Length (size) and Position.

Example

```
1.    using System;
2.    using System.IO;
3.    public class StreamFileIO {
4.          public static void Main() {
5.             try {
6.                  FileStream inFile = new FileStream("saudiflag.gif", FileMode.Open);
7.                  FileStream outFile = new FileStream("flagcopy.gif", FileMode.Create);
8.                  byte[] buffer = new byte[1024];
9.
10.                 while (inFile.Position < inFile.Length) {
11.                          int read = inFile.Read(buffer, 0, buffer.Length);
12.                          outFile.Write(buffer, 0, read);
13.                 }
14.                 inFile.Close();
15.                 outFile.Close();
16.            }
17.            catch (FileNotFoundException) {
18.                 Console.WriteLine("Sorry, File not found");
19.            }
20.            catch (Exception e) {
21.                 Console.WriteLine("Sorry, Exception: "+e);
22.            }
23.
24.         }
25.   }
```

Text IO

- For the purpose of Text IO, C# has separate classes for input and output, namely, StreamReader and SreamWriter, respectively.
- SreamReader most common constructors:

```
public StreamReader(string path)  //using UTF-8 as the default encoding scheme.
public StreamReader(string path, Encoding encoding)
```

- Encoding types
  - System.Text.ASCIIEncoding, System.Text.UnicodeEncoding, System.Text.UTF7Encoding, System.Text.UTF8Encoding
- Methods

int Read()	Reads a single character, returns –1 if end of stream
int Peek()	Returns the next character without reading it, or –1 if end of stream.
void Read(char[ ], int offset, int count)	Reads an array of characters
string ReadLine()	Reads a line of characters
string ReadToEnd()	Reads from the current position to end

StreamWriter

- Common constructors

public StreamWriter(string path) //using UTF-8 as the default encoding scheme.

public StreamWriter(string path, bool append) //for appending

public StreamWriter(string path, Encoding encoding)

public StreamWriter(string path, bool append, Encoding encoding) //for appending

- Basic methods of the StreamWriter class
  - Write and WriteLine.
    - These are overloaded to accept char, char[], string, and each of the primitive types.

Example

```
1.    using System;
2.    using System.IO;
3.    public class TextFileIO {
4.          public static void Main() {
5.             try {
6.                  StreamReader inFile = new StreamReader("SWE344.txt");
7.                  StreamWriter outFile = new StreamWriter("output.txt");
8.
9.                  String line = null;
10.                 while ((line = inFile.ReadLine()) != null) {
11.                           Console.WriteLine(line);
12.                           outFile.WriteLine(line);
13.                 }
14.                 inFile.Close();
15.                 outFile.Close();
16.            }
17.            catch (FileNotFoundException) {
18.                 Console.WriteLine("Sorry, File not found");
19.            }
20.            catch (Exception e) {
21.                 Console.WriteLine("Sorry, Exception: "+e);
22.            }
23.         }
24.   }
```

File Handling

- C# provides a number of classes for handling Files at the file system level. Operations such as Create, Copy, Move, Delete for both files and directories are provided through the following classes:
  - File, FileInfo, Directrory and DirectoryInfo.
- The File class
  - All the methods in the File class are static, so there is no constructor.
  - A problem with the File class is that each time one of its methods is called, the system must check that the user has permission on the file system before such operation is allowed. This can lead to inefficiency if there is frequent calls to the methods.
  - To solve this problem, C# provides the FileInfo class with similar set of methods, but which are non-static. In this case, permission is only checked at the point of creating an instance of FileInfo.
- The Directory class provides static methods similar to those of File class, but for manipulating directories. The DrectoryInfo class provides instance methods, which are more efficient.

File Class

static FileStream Create(String path)	Creates the file specified by path and returns its FileStream which can be used to write streams to the file.
static StreamWriter CreateText(string path)	Creates the file specified by path, and returns a StreamWriter which can be used to write text to the file.
static StreamWriter AppendText(string path)	Opens a file for appending text.
static FileStream Open(string path, FileMode mode)	opens a FileStream on the file specified by path using one of the FileStream open methods described earlier.
static FileStream Open(string path, FileMode mode, FileAccess access)	
static FileStream Open(string path, FileMode mode, FileAccess access, FileShare share)	
static StreamReader OpenText(string path)	Opens a file for text input.
static void Copy(string source, string destination)	Copies source file as destination
static void Move(string source, string destination)	Moves source file to destination
static void Delete(string path)	Deletes the file specified by path
static bool Exists(string path)	Checks if the file specified by path exists

Example

- shows how to use the methods of the File class.

```
1.   using System;
2.   using System.IO;
3.   public class CopyFile {
4.        public static void Main() {
5.           try {
6.                File.Copy("saudiflag.gif", "saudiflag2.gif");
7.           }
8.           catch (FileNotFoundException) {
9.                Console.WriteLine("Sorry, File not found");
10.          }
11.          catch (Exception e) {
12.               Console.WriteLine("Sorry, Exception: "+e);
13.          }
14.       }
15.  }
```