# Exception & Utility Classes

# Agenda

- Exception Handling
- Random Number
- Enumerate
- Strong-Typed / Generic Type
- Generic Collection
- Using Regular Expression
- File I/O
- Encryption Functions

# Exception Handling

# Exception Handling Introduction

- Exceptions
  - ✓ Indicate problems that occur during a program's execution
  - ✓ Occur infrequently
- Exception handling
  - ✓ Can resolve exceptions
    - Allow a program to continue executing or
    - Notify the user of the problem and
    - Terminate the program in a controlled manner
  - ✓ Makes programs robust and fault-tolerant

# General form of try-catch-finally

- If any exception occurs inside the try block, the control transfers to the appropriate catch block and later to the finally block.
- But in C#, both catch and finally blocks are optional.
- The try block can exist either with one or more catch blocks or a finally block or with both catch and finally blocks.

```
try
{
        // Statement which can cause an exception.
}
catch(Type x)
{
        // Statements for handling the exception
}
finally
{
        //Any cleanup code
}
```

# Exception Handling Samples

Program will compile but will show an error during execution

```
using System;
    class MyClient
    {
        public static void Main() {
        int x = 0;
        int div = 100/x;
        Console.WriteLine(div);
       }
    }
```

# Exception Handling Samples

## With Exception Handling

```
public static void Main() {
  int x = 0;
  int div = 0;
  try {
    div = 100/x;
    Console.WriteLine("This line in not
executed"); }
  catch(DivideByZeroException de) {
    Console.WriteLine("Exception
occured");}
  Console.WriteLine("Result is {0}",
div);
  }
```

## Finally Block

```
public static void Main() {
  int x = 0;
  int div = 0;
  try {  div = 100/x;
         Console.WriteLine("Not
executed line");}
  catch(DivideByZeroException de) {
   Console.WriteLine("Exception
occured");}
  finally {
    Console.WriteLine("Finally
Block");  }
  Console.WriteLine("Result is {0}",
div);}
```

# Exception Handling Notes

- Catch block can be optional
- There can be multiple catch blocks
- We can handle all exceptions with the Exception object
- In C#, it is possible to throw an exception programmatically.

```
public static void Main() {
    int x = 0;
    int div = 0;

try {
    div = 100 / x;
    throw new DivideByZeroException(
                                    "Invalid Division");
}
catch(DivideByZeroException e) {
   Console.WriteLine(e);
}
Console.WriteLine("LAST STATEMENT");
}
```

# User-defined Exceptions

```
class MyException : Exception
{
    public MyException(string str)
    {
        Console.WriteLine("User defined exception");
    }
}
```

In C#, it is possible to create our own exception class. But Exception must be the ultimate base class for all exceptions in C#. So the user-defined exception classes must inherit from either Exception class or one of its standard derived classes.

```
class MyClient
{
    public static void Main()
    {
        int x = 0;
        int div = 0;
        try
        {
            div = 100 / x;
            throw new MyException("rajesh");
        }
        catch (Exception e)
        {
          Console.WriteLine("Exception caught here" +
        e.ToString());
        }
        Console.WriteLine("LAST STATEMENT");
    }
}
```

# Exception Handling Guidelines

- Exceptions should be used to communicate exceptional conditions.
- Don't use them to communicate events that are expected, such as reaching the end of a file.
- If there's a good predefined exception in the System namespace that describes the exception condition (one that will make sense to the users of the class) use that one rather than defining a new exception class, and put specific information in the message.

# Exception Handling Guidelines

- *Do Not Catch Exceptions That You Cannot Handle*.

```
try { intNumber = int.Parse(strNumber);}
   catch (Exception ex)
 {    Console.WriteLine("Can't convert the string to " + "a number: "
+ ex.Message);}
```

```
try { intNumber =int.Parse(strNumber);}
catch (ArgumentNullException ex)
    { Console.WriteLine(@"input is
null"); }
  catch( FormatException ex)
     { Console.WriteLine(@"Incorrect
format"); }
```

✓ You should never catch System.Exception or System.SystemException in a catch block because you could inadvertently hide run-time problems like Out Of Memory.

# Exception Handling Guidelines

- Use validation code to avoid unnecessary exceptions.

```
double result = 0;
 try{
 result = numerator/divisor; }
 catch( System.Exception e)
{ result = System.Double.NaN; }
```

more efficient.

```
double result = 0;
if ( divisor != 0 ) result = numerator/divisor;
 else result = System.Double.NaN;
```

# Exception Handling Guidelines

- Do Not Use Exceptions to Control Application Flow

```
static void ProductExists( string ProductId)
{ //... search for Product
if ( dr.Read(ProductId) ==0 )  // no record found, ask to create
{ throw( new Exception("Product Not found")); } }
```

```
static bool ProductExists( string ProductId)
 { //... search for Product
 if ( dr.Read(ProductId) ==0 ) // no record found, ask to create
{ return false; } . . .
}
```

# Exception Handling Guidelines

- The following code ensures that the connection is always closed.

```
SqlConnection conn = new SqlConnection("...");
try { conn.Open();
//.Do some operation that might cause an exception
// Calling Close as early as possible conn.Close();
 // ... other potentially long operations
 } Finally
 { if (conn.State==ConnectionState.Open) conn.Close(); //
   ensure that the connection is closed }
```

# Exception Handling Guidelines

- The cost of using throw to rethrow an existing exception is approximately the same as throwing a new exception. In the following code, there is no savings from rethrowing the existing exception.

```
try { // do something that may throw an exception}(Exceptione)
   catch { // do something with e throw; }
```

- Do not catch exceptions that you do not know how to handle and then fail to propagate the exception

```
try { // exception generating code
    } catch(Exception e)
   {// Do nothing }
```

# Random Number

```
using System.Random;

Random rdm = new Random();
int i = rdm.Next(10, 100);// A random integer between 10 and 99
i = rdm.Next(100);          // Equivalent Next(0, 100)
i = rdm.Next();             // Equivalent Next(0, Int32.MaxValue)
double d = rdm.NextDouble();// A random double greater or equal
                            // zero and less than 1.0
byte[] bar = new byte[10];
rdm.NextBytes(bar);         // an array of byte numbers
```

# Enumerate

```
// Default starts from Zero
enum WorkingDays {Monday, Tuesday, Wednesday, Thursday,
Friday};
int i = (int)WorkingDays.Monday; // i = 0
// Assigned value
enum WorkingDays {…, Wednesday = 5, …};
int i = (int)WorkingDays.Friday; // i = 7
// Using
WorkingDays wd = WorkingDays.Tuesday;
switch (wd){…}
string n = Enum.GetName(typeof(WorkingDays), 6);
       n = wd.ToString();          // n = "Tuesday"
```

# Strong-Typed / Generic Type

```
class GenericType<T>{
   // T is a type representation, not a specific type
   public T PropertyT(get; set;}
                    }
class A{}
// use generic class with specific type int
GenericType<int> gInt = new GenericType<int>();
gInt.PropertyT = 5;
int i = PropertyT;
// use generic class with specific type A
GenericType<A> gA = new GenericType<A>();
gA.PropertyT = new A();
A a = gA.PropertyT;
```

# Multi type Param & Type Param Constraint

```
class GenericType<T, U, V>{
    // Anny positive number of type
     parameter,
    private T aT;
    private U aU;
    private V aV;
    …
}
```

```
class GenericType<T> where T:A{
    // A is a specific type
    private T aT; …
}
class A {}
class B:A {}
class C {}
GenericType<A> gA = new GenericType<A>();
// OK
GenericType<B> gB = new GenericType<B>();
// OK too
GenericType<C> gA = new GenericType<C>();
// Error, C is not A
```

# Generic Collection

# Generic Collection

- Array
- List<T>
- Dictionary<TKey, Tvalue>

# System.Array class

```
public class A:IComparable<A>{
// implements IComparable for sorting
   public int i{get;set;}
   public int CompareTo(A another){
   if (i == another.i) return 0;
       if (i < another.i) return -1;
       return 1;
   }
}
A[] ar = new A[10];
int i = ar.Length;                  // 10
for (i = 10; i > 0; i--){           // Initialize
   A ai = new A(); ai.i = i;
   ar[10 - i] = ai;                 // access by index
}                                   // 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
```

# Array class: Operation

```
Array.Sort(ar);                     // 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

A a = ar[3];                           // 4
i = Array.IndexOf(ar, a);              // 3
i = Array.IndexOf(ar, a, 1);           // 3
i = Array.IndexOf(ar, a, 2, 1);        // -1, not found
                                    // last number is search section length
i = Array.IndexOf(ar, a, 2, 2);        // 3,

ar[6] = a;                          // 1, 2, 3, 4, 5, 6, 4, 8, 9, 10
i = Array.IndexOf(ar, a, 4, 5);        // 6
i = Array.LastIndexOf(ar, a);          // 6
Array.Reverse(ar);                  // 10, 9, 8, 4, 6, 5, 4, 3, 2, 1
```

# System.Collections.Generic.List

```
public class A:IComparable<A>{…}

List<A> al = new List<A>();
int i = ar.Count;                    // 0

for (i = 10; i > 0; i--){            // Initialize
   A ai = new A(); ai.i = i;
   al.Add(ai);
}                                    // 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
…
```

# List class: Operation

```
al.Sort();                          // 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
A a = al[3];                        // 4, access by index
i = al.IndexOf(a);                  // 3
i = al.IndexOf(a, 1);               // 3
i = al.IndexOf(a, 2, 1);            // -1, not found
                                 // last number is search section
   length
i = al.IndexOf(a, 2, 2);            // 3,
ar[6] = a;                          // 1, 2, 3, 4, 5, 6, 4, 8, 9, 10
i = al.IndexOf(a, 4, 5);            // 6
i = al.LastIndexOf(a);              // 6
al.Reverse();                       // 10, 9, 8, 4, 6, 5, 4, 3, 2, 1
al.RemoveAt(5);                     // 10, 9, 8, 4, 6, 4, 3, 2, 1
```

# System.Collections.Generic.Dictionnary

```
class A{public int i{get; set;}}
Dictionary<int, A> ad = new Dictionary<int, A>();
   // use the hash code of key then no order is warranted
int i = ad.Count;                  // 0

A aA = new A(); aA.i = 1;
ad.Add(1, aA);                     // add new
i = ad.Count;                      // 1
ad.Add(1, aA);                     // Error, key existed
bool b = ad.ContainsKey(1);        // true
A oA = ad[1]                       // access by key like array
b = oA == aA                       // true
b = ad.ContainsValue(aA);          // true
b = ad.Remove(2);                  // false
b = ad.Remove(1);                  // access by key
```

# Using Regular Expression

```
using System.Text.RegularExpressions;

string emailpattern = @"\w+(.\w+)*@\w+.(\w{2,})(.\w{2,})*";
Regex regx = new Regex(emailpattern);

string email = "fwa.ctc@fsoft.com.vn";
bool b = regx.IsMatch(email);    // Ignore the position
Match m = regx.Match(email);     // get more information
b = m.Success;                   // true
int i = m.Index;               // the position of the 1st matched character
string textfound = m.Value;      // "fwa.ctc@fsoft.com.vn" in this case

email = "fwa@fsoft.com.vn, fpt@fsoft.com.vn";
MatchCollection ms = regx.Matches(email); // multiple results
foreach (Match m1 in ms){
   i         = m1.Index;         // 0 then 18
   textfound = m1.Value;         // fwa@fsoft.com.vn then fpt@fsoft.com.vn
                                 // m1.Success is always true in collection
}
// Removes whitespace between a word character and . or ,
Regex rgx = new Regex(@"(\w)(\s+)([.,])");
string s = rgx.Replace("sdfd . sdfgdg ,", @"$1$3");
```

# LINQ for Object

```
// Specify the data source.
int[] scores = {97, 92, 81, 60};

// Define the query expression.
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query.
foreach (int i in scoreQuery){
    Console.Write(i + " ");
}
// Output: 97 92 81
```

# File I/O

# File I/O Why read or write to the file system?

- Show existing data to user
- Integrate user-provided data
- Serialize objects out of memory
- Persist data across sessions
- Determine environment configuration

# How do we write to files?

- This is simplified with Framework methods; open / shut
  - ✓ File.WriteAllText / ReadAllText
- Open for reading to keep open and keep writing
- Open as stream for large payloads and realtime processing

# Plan text file I/O

```
// Type
using System.IO.StreamReader;
using System.IO.StreamWriter;
try{
   // File exist:
   if (File.Exists("a.txt")){
      // Open
      StreamReader input = new StreamReader("a.txt");
      StreamWriter output = new StreamWriter ("b.txt");
      // Repeat access until end of input
      string line;
      while ((line = input.ReadLine()) == null){
         output.WriteLine(line); }
      //Close
      input.Close(); output.Close();
   }
} catch (IOException e){
   System.Console.WriteLine(e.Message);}
```

# File IO Special folder

```
// special folders
var docs = Environment.SpecialFolder.MyDocuments;
var app = Environment.SpecialFolder.CommonApplicationData;
var prog = Environment.SpecialFolder.ProgramFiles;
var desk = Environment.SpecialFolder.Desktop;

// application folder
var dir = System.IO.Directory.GetCurrentDirectory();

// isolated storage folder(s)
var iso = IsolatedStorageFile
    .GetStore(IsolatedStorageScope.Assembly, "Demo")
    .GetDirectoryNames("*");

// manual path
var temp = new System.IO.DirectoryInfo("c:\temp");
```

# Files system accessing

```
// files
foreach (var item in System.IO.Directory.GetFiles(dir))
    Console.WriteLine(System.IO.Path.GetFileName(item));

// rename / move
var path1 = "c:\temp\file1.txt";
var path2 = "c:\temp\file2.txt";
System.IO.File.Move(path1, path2);

// file info
var info = new System.IO.FileInfo(path1);
Console.WriteLine("{0}kb", info.Length / 1000);
```

# Encryption

# What is encryption

- An encryption algorithm makes data unreadable to any person  or system until the associated decryption algorithm is applied.
  - ✓ Encryption does not hide data; it makes it unreadable
  - ✓ Encryption is not the same as compression
- Types of encryption
  - ✓ File Encryption
  - ✓ Windows Data Protection
  - ✓ Hashing, used for signing and validating
  - ✓ Symmetric and Asymmetric

# Encryption Methods

- File Encryption
  - ✓ Encrypts and decrypts files
  - ✓ Fast to encrypt/decrypt
  - ✓ Based on user credentials
- Windows Data Protection
  - ✓ Encrypts and decrypts byte[]
  - ✓ Fast to encrypt/decrypt
    Based on user credentials

# Hashing DEMO

- One-way encryption
- Common algorithms:
  - ✓ MD5 (generates a 16 character hash than can be stored in a Guid)
  - ✓ SHA (SHA1, SHA256, SHA384, SHA512)
- Fast (depending on chosen algorithm)
- Used for storing passwords, comparing files, data corruption/tamper checking
  - ✓ Use SHA256 or greater for passwords or other sensitive data

# Symmetric Encryption DEMO

- One key is used for both encryption and decryption
- Faster than asymmetric encryption
- Cryptography namespace includes five symmetric algorithms:
  - ✓ Aes (recommended)
  - ✓ DES
  - ✓ RC2
  - ✓ Rndael
  - ✓ TripeDES

# Asymmetric/Public Key Encryption DEMO

- One key is used for encryption and another key for decryption
- Commonly used for digital signatures
- Cryptography namespace includes four asymmetric algorithms:
  - ✓ DSA
  - ✓ ECDiffieHellman
  - ✓ ECDsa
  - ✓ RSA (most popular)


Questions
Answers

# Thank you