

Lab Assignment

Sudhanshu Kumar
2017IEN43
Department of Computer Science

April 22, 2020

Explain the concept of boxing and unboxing in C# with an example.

C# Type System contains three Types, they are *Value Types*, *Reference Types* and *Pointer Types*. In C# it is possible to convert a value of one type into a value of another type. C# allows us to convert a Value Type to a Reference Type, and back again to Value Types. The operation of converting a *Value Type* to a *Reference Type* is called *Boxing* and the reverse operation is called *Unboxing*.

```
int Val = 1;
Object Obj = Val;           //Boxing
int i = (int)Obj;           //Unboxing
```

From the above operation (Object Obj = i) we saw converting a value of a Value Type into a value of a corresponding Reference Type. These types of operation is called *Boxing*. The next line (int i = (int) Obj) shows extracts the Value Type from the Object. That is converting a value of a Reference Type into a value of a Value Type. This operation is called *Unboxing*.

Boxing and *Unboxing* are computationally expensive processes. When a value type is boxed, an entirely new object must be allocated and constructed, also the cast required for unboxing is also expensive computationally.

Explain the basic program structure in VB.NET.

A VB.Net program basically consists of the following parts —

- Namespace declaration
- A class or module
- One or more procedures
- Variables
- The Main procedure
- Statements & Expressions
- Comments

Let us look at a simple code that would print the words "Hello World" —

```
Imports System

Module Program
    'This program displays Hello World!
    Sub Main(args As String())
        Console.WriteLine("HelloWorld!")
        Console.ReadKey()
    End Sub
End Module
```

When the above code is compiled and executed, it produces the following result —

Hello World!

Let us look various parts of the above program —

- The first line of the program **Imports System** is used to include the *System* namespace in the program.
 - The next line has a Module declaration, the module **Program**. VB.Net is completely object oriented, so every program must contain a module of a class that contains the data and procedures that your program uses.
 - Classes or Modules generally would contain more than one procedure. Procedures contain the executable code, or in other words, they define the behavior of the class. A procedure could be any of the following —
 - Function
 - Sub
 - Operator
 - Get
 - Set
 - AddHandler
 - RemoveHandler
 - RaiseEvent
 - The next line (*'This program*) will be ignored by the compiler and it has been put to add additional comments in the program.
 - The next line defines the **Main** procedure, which is the entry point for all VB.Net programs. The *Main* procedure states what the module or class will do when executed.
 - The *Main* procedure specifies its behavior with the statement.
 - **Console.WriteLine("Hello World!")** *WriteLine* is a method of the Console class defined in the *System* namespace. This statement causes the message *"Hello World!"* to be displayed on the screen.
 - The last line **Console.ReadKey()** is for the VS.NET Users. This will prevent the screen from running and closing quickly when the program is launched from Visual Studio .NET.
-

Explain the following related to VB.NET programming.

- Data types and declaring a variable.**
- Directives**
- Creating a sub procedure.**
- Creating objects and classes**

Data types and declaring a variable

Data types refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

The following table shows some of the data types available —

Data Type	Storage Allocation
Byte	1 Byte
Char	2 Bytes
Date	8 Bytes
Decimal	16 Bytes
Double	8 Bytes
Integer	4 Bytes
Long	8 Bytes
Short	2 Bytes

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in VB.Net has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable. The **Dim** statement is used for variable declaration and storage allocation for one or more variables. The *Dim* statement is used at module, class, structure, procedure or block level. The following codes demonstrates declaration of some of the types —

```
Dim b As Byte
Dim c As Char
Dim StudentID As Integer
Dim StudentName As String
Dim Salary As Double
Dim count1, count2 As Integer
Dim status As Boolean
Dim exitButton As New System.Windows.Forms.Button
Dim lastTime, nextTime As Date
```

You can initialize a variable at the time of declaration as follows —

```
Dim StudentID As Integer = 100
Dim StudentName As String = "BillSmith"
```

Directives

The VB.Net compiler directives give instructions to the compiler to preprocess the information before actual compilation starts. All these directives begin with #, and only white-space characters may appear before a directive on a line. These directives are not statements. VB.Net compiler does not have a separate preprocessor; however, the directives are processed as if there was one. In VB.Net, the compiler directives are used to help in conditional compilation.

VB.Net provides the following set of compiler directives —

- **The #Const Directive**

This directive defines conditional compiler constants.

```
#Const state = "WEST BENGAL"
```

- **The #ExternalSource Directive**

This directive is used for indicating a mapping between specific lines of source code and

text external to the source. It is used only by the compiler and the debugger has no effect on code compilation. This directive allows including external code from an external code file into a source code file.

```
#ExternalSource("c:\vbprogs\directives.vb", 5)
Console.WriteLine("This is External Code. ")
#End ExternalSource
```

- **The #If...Then...#Else Directives**

This directive conditionally compiles selected blocks of Visual Basic code.

```
#Const TargetOS = "Linux"
#If TargetOS = "Windows 7" Then
' Windows 7 specific code
#ElseIf TargetOS = "WinXP" Then
' Windows XP specific code
#Else
' Code for other OS
#End if
```

- **The #Region Directive**

This directive helps in collapsing and hiding sections of code in Visual Basic files.

```
#Region "StatsFunctions"
' Insert code for the Statistical functions here.
#End Region
```

Creating a sub procedure

Sub procedures are procedures that do not return any value. We have been using the Sub procedure Main in all our examples. When the applications start, the control goes to the Main Sub procedure, and it in turn, runs any other statements constituting the body of the program. The Sub statement is used to declare the name, parameter and the body of a sub procedure. The syntax for the Sub statement is —

```
[Modifiers] Sub SubName [(ParameterList)]
[Statements]
End Sub
```

Where,

- **Modifiers** specify the access level of the procedure; possible values are - *Public, Private, Protected, Friend, Protected Friend* and information regarding *overloading, overriding, sharing* and *shadowing*.
- **SubName** indicates the name of the Sub.
- **ParameterList** specifies the list of the parameters.

Example—

```

Sub CalculatePay(ByRef hours As Double, ByRef wage As Decimal)
'local variable declaration
Dim pay As Double
pay = hours * wage
Console.WriteLine("Total Pay: {0:C}", pay)
End Sub

```

Creating objects and classes

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object. Objects are instances of a class. The methods and variables that constitute a class are called members of the class.

A typical declaration of class is —

```

Class Box
Public length As Double      ' Length of a box
Public breadth As Double     ' Breadth of a box
Public height As Double      ' Height of a box

Public Sub setLength(ByVal len As Double)
length = len
End Sub

Public Sub setBreadth(ByVal bre As Double)
breadth = bre
End Sub

Public Sub setHeight(ByVal hei As Double)
height = hei
End Sub

Public Function getVolume() As Double
Return length * breadth * height
End Function

End Class

```

Objects can be declared as —

```

Dim Box1 As Box = New Box()      ' Declare Box1 of type Box
Dim Box2 As Box = New Box()      ' Declare Box2 of type Box

```

Briefly discuss the 6 types of access modifiers supported in C# with an example program.

Access Modifiers (Access Specifiers) describes as the scope of accessibility of an Object and its members. All C# types and type members have an accessibility level . We can control the

scope of the member object of a class using access specifiers. We are using access modifiers for providing security of our applications. When we specify the accessibility of a type or member we have to declare it by using any of the access modifiers provided by C# language.

C# provide six access modifiers , they are as follows —

- **Private Access Modifier**

The scope of the accessibility is limited only inside the *classes* or *struct* in which they are declared. The private members cannot be accessed outside the class and it is the least permissive access level.

Example —

```
using System;
namespace ConsoleApplication1
{
    class Program
    {
        private string msg = "This variable is private ";
        private void disp(string msg)
        {
            Console.WriteLine("This function is private : " + msg);
        }
        static void Main(string[] args)
        {
            Program pr = new Program();
            Console.WriteLine(pr.msg); //Accessing private variable inside the class
            pr.disp("Hello !!"); //Accessing private function inside the class
        }
    }
}
```

Output —

```
This variable is private
This function is private : Hello !!
```

- **Public Access Modifier**

This is the most common access specifier in C# . It can be accessed from anywhere, that means there is no restriction on accessibility. The scope of the accessibility is inside *class* as well as outside. The type or member can be accessed by any other code in the same assembly or another assembly that references it.

Example —

```
using System;
namespace ConsoleApplication1
{
    class PublicAccess
    {
        public string msg = "This variable is public";
        public void disp(string msg)
        {
            Console.WriteLine("This function is public : " + msg);
        }
    }
}
```

```

    Console.WriteLine("This function is public : " + msg);
}
}
class Program
{
    static void Main(string[] args)
    {
        PublicAccess pAccess = new PublicAccess();
        Console.WriteLine(pAccess.msg); // Accessing public variable
        pAccess.disp("Hello !!"); // Accessing public function
    }
}
}

```

Output —

```

    This variable is public
    This function is public : Hello !!

```

- **Protected Access Modifier**

The scope of accessibility is limited within the *class* or *struct* and the class derived from this class.

Example —

```

using System;
namespace ConsoleApplication1
{
    class ProtectedAccess
    {
        protected string msg = "This variable is protected ";
        protected void disp(string msg)
        {
            Console.WriteLine("This function is protected : " + msg);
        }
    }
    class Program : ProtectedAccess //Derived class
    {
        static void Main(string[] args)
        {
            Program pr = new Program();
            Console.WriteLine(pr.msg); // Accessing protected variable
            pr.disp("Hello !!"); // Accessing protected function
        }
    }
}

```

- **Internal Access Modifier**

The internal access modifiers can access within the program that contain its declarations and also access only within files in the same assembly level but not from another assembly.

Example —


```

using System;
namespace ConsoleApplication1
{
    class InternalAccess
    {
        internal string msg = "This variable is internal";
        internal void disp(string msg)
        {
            Console.WriteLine("This function is internal : " + msg);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            InternalAccess iAccess = new InternalAccess();
            Console.WriteLine(iAccess.msg); // Accessing internal variable
            iAccess.disp("Hello !!"); // Accessing internal function
            Console.ReadKey();
        }
    }
}

```

Output —

```

This variable is internal
This function is internal : Hello !!

```

- **Protected Internal Access Modifier**

Protected internal is the same access levels of both protected and internal . It can access anywhere in the same assembly also be accessed within a derived class in another assembly.

Example —

```

using System;
namespace ConsoleApplication1
{
    class InternalAccess
    {
        protected internal string msg = "This variable is protected internal";
        protected internal void disp(string msg)
        {
            Console.WriteLine("This function is protected internal : " + msg);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            InternalAccess iAccess = new InternalAccess();
            Console.WriteLine(iAccess.msg); // Accessing internal variable
        }
    }
}

```

```

iAccess.disp("Hello !!"); // Accessing internal function
Console.ReadKey();
}
}
}

```

Output —

```

This variable is  protected internal
This function is protected internal : Hello !!

```

- **Private Protected Access Modifier**

The new modifier private protected really means *protected and also internal*. That is, member is accessible only to child classes which are in the same assembly, but not to child classes which are outside assembly. That is useful if you build hierarchy of classes in your assembly and do not want any child classes from other assemblies to access certain parts of that hierarchy.

Example —

```

Assembly1.cs
// Compile with: /target:library
public class MyBaseClass
{
    private protected int count = 0;
}
public class MyDerivedClass1 : MyBaseClass
{
    void callMe()
    {
        MyBaseClass obj = new MyBaseClass();
        obj.count = 5;
        // Error CS1540, because the variable count can only be accessed by
        // classes derived from MyBaseClass.
        count = 5; // OK, accessed through the current derived class instance
    }
}

```

```

Assembly2.cs
// Compile with: /reference:Assembly1.dll
class MyDerivedClass2 : MyBaseClass
{
    void callMe()
    {
        count = 10;
        // Error CS0122, because count can only be
        // accessed by types in Assembly1
    }
}

```

- **Default access modifiers in C#**

When no access modifier is set, a default access modifier is used. So there is always some form of access modifier even if it's not set. The default access for everything in C# is "the most restricted access you could declare for that member".

- **Static modifier**

The static modifier on a class means that the class cannot be instantiated, and that all of its members are static. A static member has one version regardless of how many instances of its enclosing type are created.

Explain any 4 types of C# collections with an example.

i. ArrayList

- The Add() method of ArrayList is used to add new items in ArrayList.

```
ArrayList.Add(object)
```

- ArrayList Insert(Int32, Object) method inserts an element into the ArrayList at the specified index.

```
ArrayList.Insert(index,object)
```

- ArrayList remove() method removes the first occurrence of a specific object from the ArrayList.

```
ArrayList.Remove(object)
```

- *ArrayList.RemoveAt(index)* remove the specified index element from arraylist.

- *ArrayList.RemoveRange(Int32, Int32)* method is used to remove a range of elements from the ArrayList.

- ArrayList Sort() method sorts the elements in the entire ArrayList.

```
ArrayList.Sort()
```

```
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arrList = new ArrayList();
            arrList.Add("North");
            arrList.Add("West");
            arrList.Add("South");
            Console.WriteLine("ArrayList Elements...");
            foreach (object item in arrList)
```

```

{
Console.WriteLine(item);
}

//insert an item at the top of arraylist
arrList.Insert(0, "Directions");
//insert an item at the 3rd position of arraylist
arrList.Insert(2, "East");
Console.WriteLine("ArrayList Elements after insert...");
foreach (object item in arrList)
{
Console.WriteLine(item);
}

//remove() method removes the first occurrence
arrList.Remove("West");
Console.WriteLine("ArrayList Elements after remove()...");
foreach (object item in arrList)
{
Console.WriteLine(item);
}
arrList.Insert(3,"West");
//remove element at index 1
arrList.RemoveAt(1);
Console.WriteLine("ArrayList Elements after removeAt()...");
foreach (object item in arrList)
{
Console.WriteLine(item);
}
arrList.Insert(1,"North");

//remove range of elements from arraylist
arrList.RemoveRange(1,2);
Console.WriteLine("ArrayList Elements after RemoveRange()...");
foreach (object item in arrList)
{
Console.WriteLine(item);
}
arrList.Insert(1,"North");
arrList.Insert(2,"East");

//sort arraylist elements
arrList.Sort();
Console.WriteLine("ArrayList Elements after sort()...");
foreach (object item in arrList)
{
Console.WriteLine(item);
}
Console.ReadKey();
}
}

```

```
}
```

ii. **HashTable**

Hashtable in C# represents a collection of key/value pairs which maps keys to value. Any non-null object can be used as a key but a value can. We can retrieve items from hashtable to provide the key. Both keys and values are Objects.

The commonly used functions in Hashtable are —

- *Add* : To add a pair of value in Hashtable

```
HashTable.Add(Key, Value)
```

- *ContainsKey* : Check if a specified key exist or not

```
bool HashTable.ContainsKey(key)
```

- *ContainsValue* : Check the specified Value exist in Hashtable

```
bool HashTable.ContainsValue(Value)
```

- *Remove* : Remove the specified Key and corresponding Value

```
HashTable.Remove(Key)
```

```
namespace WindowsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            Hashtable weeks = new Hashtable();
            weeks.Add("1", "SunDay");
            weeks.Add("2", "MonDay");
            weeks.Add("3", "TueDay");
            weeks.Add("4", "WedDay");
            weeks.Add("5", "ThuDay");
            weeks.Add("6", "FriDay");
            weeks.Add("7", "SatDay");
            //Display a single Item

            MessageBox.Show(weeks["5"].ToString ());
            //Search an Item
            if (weeks.ContainsValue("TueDay"))
            {
                MessageBox.Show("Find");
            }
        }
    }
}
```

```

    }
    else
    {
        MessageBox.Show("Not find");
    }
    //remove an Item
    weeks.Remove("3");
    //Display all key value pairs
    foreach (DictionaryEntry day in weeks )
    {
        MessageBox.Show (day.Key + " " + day.Value );
    }
}
}
}
}

```

iii. Stack

The *Stack class* represents a last-in-first-out (LIFO) Stack of Objects. Stack is implemented as a circular buffer. It follows the Last In First Out (LIFO) system. That is we can push the items into a stack and get it in reverse order. Stack returns the last item first. As elements are added to a Stack, the capacity is automatically increased as required through reallocation.

Commonly used methods —

- *Push* : Add (Push) an item in the Stack data structure

```
Stack.Push(Object)
```

- *Pop* : Pop return the last Item from the Stack

```
Object Stack.Pop()
```

- *Contains* : Check the object contains in the Stack

```
Stack.Contains(Object)
```

```

using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)

```

```

{
Stack days = new Stack();
days.Push("SunDay");
days.Push("MonDay");
days.Push("TueDay");
days.Push("WedDay");
days.Push("ThuDay");
days.Push("FriDay");
days.Push("SaturDay");
if (days.Count ==7)
{
MessageBox.Show(days.Pop().ToString ());
}
else
{
MessageBox.Show("SaturDay does not exist");
}
}
}
}
}

```

iv. Queue

The Queue works like FIFO system , a first-in, first-out collection of Objects. Objects stored in a Queue are inserted at one end and removed from the other. The Queue provide additional insertion, extraction, and inspection operations. We can Enqueue (add) items in Queue and we can Dequeue (remove from Queue) or we can Peek (that is we will get the reference of first item) item from Queue. Queue accepts null reference as a valid value and allows duplicate elements.

Some important functions in the Queue Class are follows :

- *Enqueue* : Add an Item in Queue
 Queue.Enqueue(Object)
- *Dequeue* : Remove the oldest item from Queue
 Object Queue.Dequeue()
- *Peek* : Get the reference of the oldest item
 Object Queue.Peek()

```

using System;
using System.Collections;
using System.Windows.Forms;

namespace WindowsApplication1
{
public partial class Form1 : Form
{
public Form1()

```

```

{
InitializeComponent();
}

private void button1_Click(object sender, EventArgs e)
{
Queue days = new Queue();
days.Enqueue("Sunday");
days.Enqueue("Monday");
days.Enqueue("Tuesday");
days.Enqueue("Wednesday");
days.Enqueue("Thursday");
days.Enqueue("Friday");
days.Enqueue("Saturday");

MessageBox.Show (days.Dequeue().ToString ());

if (days.Contains("Monday"))
{
MessageBox.Show("The queue contains Monday");
}
else
{
MessageBox.Show("Does not match any entries");
}
}
}
}

```

How to create a PDF file from text file using C# ? Write a program to implement the same.

Pdfsharp is a popular open source framework which could be used to create PDF files programmatically. In many situations we need PDF documents instead of text documents because text files are listed as the simplest file format, which limits users only to edit words. From the following program you can easily convert a text file to a PDF formatted document.

```

using System;
using System.Windows.Forms;
using PdfSharp;
using PdfSharp.Drawing;
using PdfSharp.Pdf;
using System.Diagnostics;
using System.IO;

namespace textToPdf
{
public partial class Form1 : Form

```



```

{
public Form1()
{
InitializeComponent();
}

private void button1_Click(object sender, EventArgs e)
{
try
{
string line = null;
System.IO.TextReader readFile = new StreamReader("text.txt");
int yPoint = 0;

PdfDocument pdf = new PdfDocument();
pdf.Info.Title = "txt to pdf";
PdfPage pdfPage = pdf.AddPage();
XGraphics graph = XGraphics.FromPdfPage(pdfPage);
XFont font = new XFont("Verdana", 20, XFontStyle.Regular);

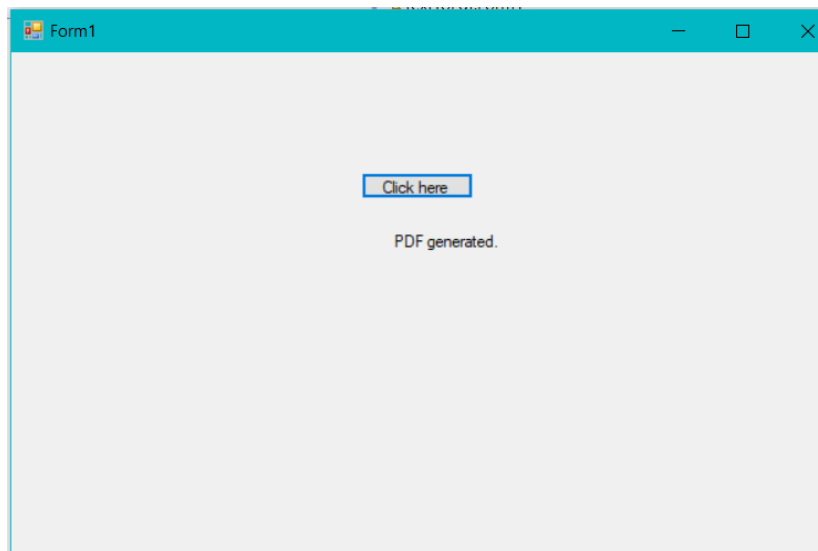
while (true)
{
line = readFile.ReadLine();
if (line == null)
{
break;
}
else
{
graph.DrawString(line, font, XBrushes.Black, new XRect(40, yPoint, pdfPage.Width.Point,
pdfPage.Height.Point), XStringFormats.TopLeft);
yPoint = yPoint + 40;
}
}

string pdfFilename = "texttoPDF.pdf";
pdf.Save(pdfFilename);
readFile.Close();
readFile = null;
Process.Start(pdfFilename);
label1.text = "PDF generated";
}

catch (Exception ex)
{
MessageBox.Show(ex.ToString());
}
}
}
}
}

```

Output



Write a C# program to read and import excel file into dataset using OLEDB .

```
using System;
using System.Data;
using System.Data.OleDb;
using System.Drawing;
using System.Windows.Forms;

namespace excelToDatabase
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

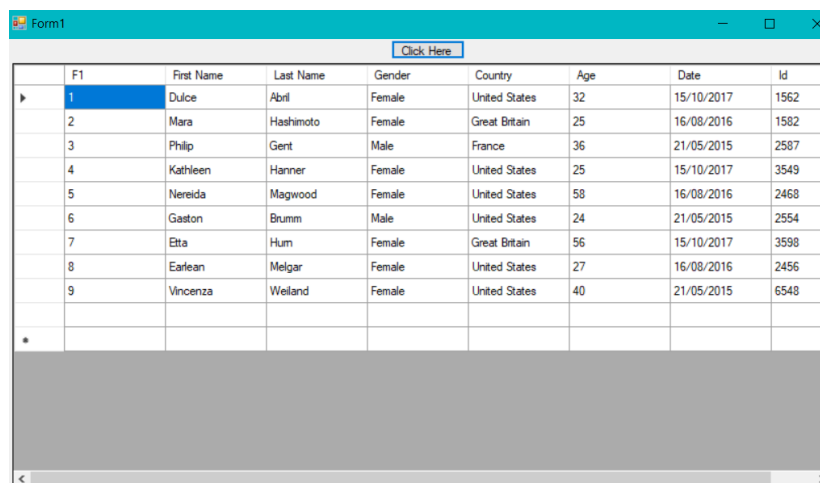
        private void button1_Click(object sender, EventArgs e)
        {
            try
            {
                OleDbConnection myConnection;
                DataSet dataSet;
                OleDbDataAdapter myCommand;
                myConnection = new OleDbConnection("provider=Microsoft.Jet.OLEDB.4.0;Data
                Source='file.xls';Extended Properties=Excel 8.0;");
                myCommand = new OleDbDataAdapter("select * from [Sheet1$]", myConnection);
                myCommand.TableMappings.Add("Table", "TestTable");
                dataSet = new DataSet();
                myCommand.Fill(dataSet);
            }
            catch { }
        }
    }
}
```

```

dataGridView1.DataSource = dataSet.Tables[0];
myConnection.Close();
}
catch(Exception ex)
{
    MessageBox.Show(ex.ToString());
}
}
}
}
}

```

Output



	F1	First Name	Last Name	Gender	Country	Age	Date	Id
▶	1	Dulce	Abril	Female	United States	32	15/10/2017	1562
	2	Mara	Hashimoto	Female	Great Britain	25	16/08/2016	1582
	3	Philip	Gent	Male	France	36	21/05/2015	2587
	4	Kathleen	Hanner	Female	United States	25	15/10/2017	3549
	5	Nereida	Magwood	Female	United States	58	16/08/2016	2468
	6	Gaston	Brumm	Male	United States	24	21/05/2015	2554
	7	Etta	Hum	Female	Great Britain	56	15/10/2017	3598
	8	Earlean	Melgar	Female	United States	27	16/08/2016	2456
	9	Vincenza	Weiland	Female	United States	40	21/05/2015	6548
*								

Write a Program to insert a data into excel file using OLEDB.

```

using System;
using System.Data;
using System.Data.OleDb;
using System.Drawing;
using System.Windows.Forms;

namespace insertData
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {

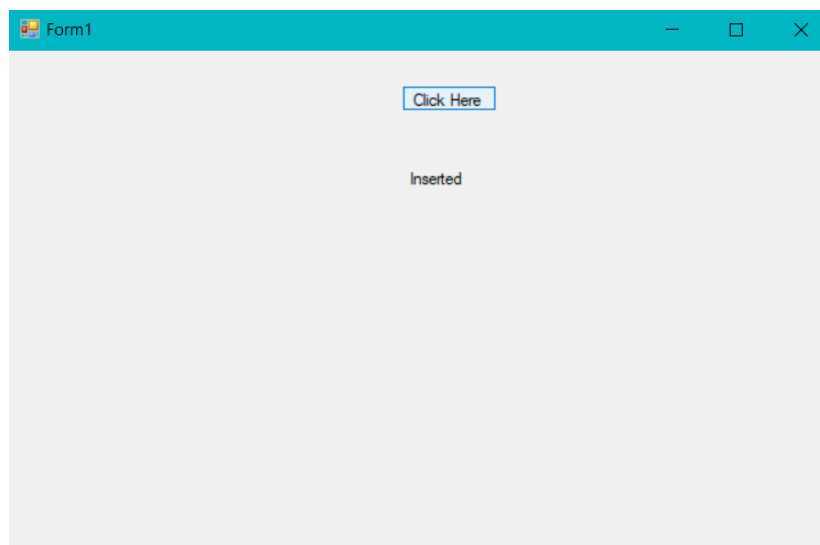
```

```

OleDbConnection myConnection;
OleDbCommand myCommand = new OleDbCommand();
string sql = null;
myConnection = new OleDbConnection("provider=Microsoft.Jet.OLEDB.4.0;Data
    Source='file.xls';Extended Properties=Excel 8.0;");
myConnection.Open();
myCommand.Connection = myConnection;
sql = "Insert into [Sheet1$] (id) values('25')";
myCommand.CommandText = sql;
myCommand.ExecuteNonQuery();
myConnection.Close();
label1.Text = "Inserted";
}
catch(Exception ex)
{
    MessageBox.Show(ex.ToString());
}
}
}
}

```

Output



Write a program to update a data in an existing excel file using OLEDB.

```

using System;
using System.Data;
using System.Data.OleDb;
using System.Drawing;
using System.Windows.Forms;

```

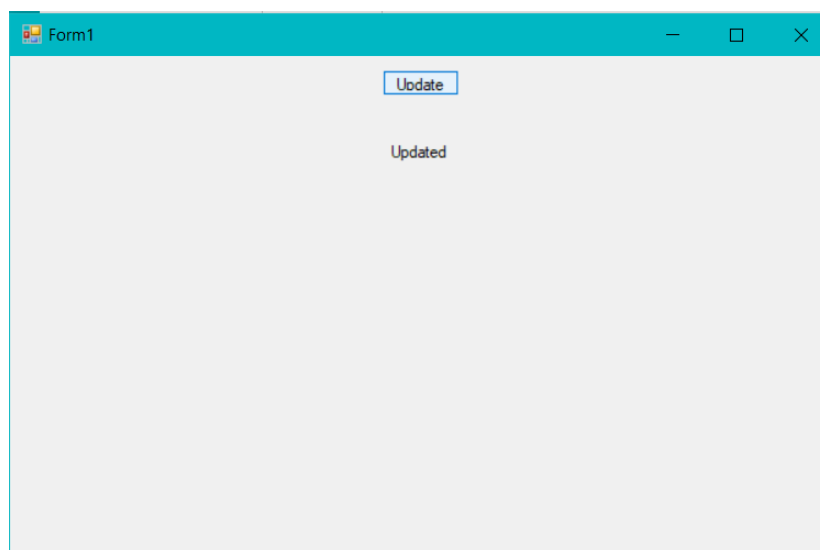
```

namespace updateData
{
public partial class Form1 : Form
{
public Form1()
{
InitializeComponent();
}

private void button1_Click(object sender, EventArgs e)
{
try
{
OleDbConnection myConnection;
OleDbCommand myCommand = new OleDbCommand();
string sql = null;
myConnection = new OleDbConnection("provider=Microsoft.Jet.OLEDB.4.0;Data
    Source='file.xls';Extended Properties=Excel 8.0;");
myConnection.Open();
myCommand.Connection = myConnection;
sql = "Update [Sheet1$] set id = 23 where id=25";
myCommand.CommandText = sql;
myCommand.ExecuteNonQuery();
myConnection.Close();
label1.Text = "Updated";
}
catch(Exception ex)
{
MessageBox.Show(ex.ToString());
}
}
}
}

```

Output



Write a program to send an email with attachment using C# forms.

```
using System.Windows.Forms;
using System.Net.Mail;
using System.Net;

namespace mail
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

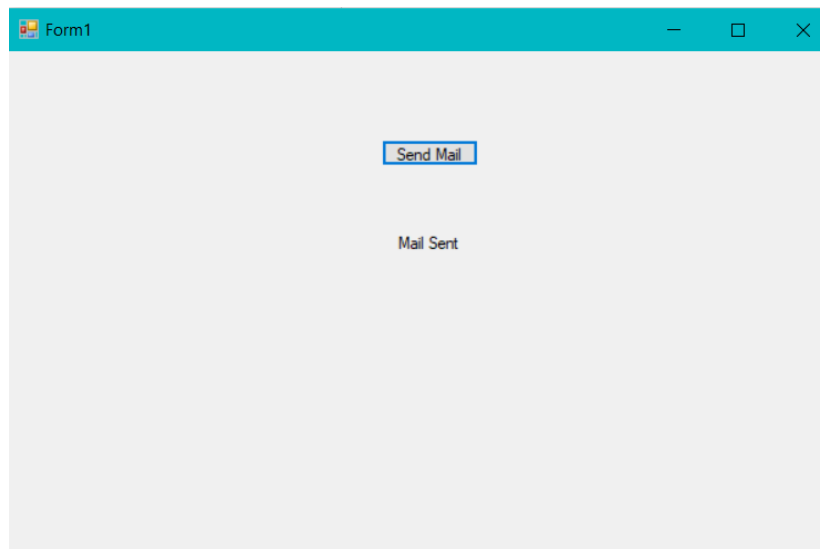
        private void button1_Click(object sender, EventArgs e)
        {
            try
            {
                MailMessage mail = new MailMessage();
                SmtpClient smtpServer = new SmtpClient("smtp.gmail.com");
                mail.From = new MailAddress("sudhiskpl@gmail.com");
                mail.To.Add("s.kumar.s23798@gmail.com");
                mail.Subject = "Test";
                mail.Body = "Please find attachment";

                Attachment attachment;
                attachment = new Attachment("c#.pdf");
                mail.Attachments.Add(attachment);

                smtpServer.Port = 587;
                smtpServer.UseDefaultCredentials = false;
                smtpServer.Credentials = new NetworkCredential("sudhiskpl@gmail.com",
                    "passwordNahiBatayenge");
                smtpServer.EnableSsl = true;
                smtpServer.Send(mail);

                label1.Text = "Mail Sent";
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.ToString());
            }
        }
    }
}
```

Output



Thanks, Stay Hydrated and Keep Breathing.