



Visual C#.NET: Console Applications and Windows Forms

Visual Studio .NET is a Microsoft-integrated development environment (IDE) that can be used for developing consoles, graphical user interfaces (GUIs) and Windows Forms. This book explores the use of C# for creating a variety of applications that run on the .NET framework. The C# syntax is explored through the use of several examples that allow the user to create applications in console mode, interact with objects in windows forms, establish connection to a relational database and explore the use of XML documents. Eight scenarios for the adoption of this technology are also exploited.

Fernando Almeida, PhD.

July, 2018

Table of Contents

Acronyms.....	4
Glossary	5
1. Introduction	6
1.1 Contextualization	6
1.2 Objectives	6
1.3 Book Structure	6
2. The basic of console applications.....	7
3. The basic of windows forms	11
4. C# syntax	15
4.1 Variables.....	15
4.2 Operators.....	17
4.3 if-else statement.....	19
4.4 Switch statement.....	20
4.5 For loop.....	20
4.6 While loop	21
4.7 Arrays	22
4.8 Arrays class	25
4.9 Functions	26
4.10 Class	28
4.11 Structs.....	31
4.12 Strings.....	33
4.13 Exceptions	35
4.14 Files	38
4.15 Collections	41
4.16 Databases.....	42
4.17 XML	44
5. Scenarios	48
5.1 Console Application: Price of products	48
5.2 Console Application: Prime numbers	48

5.3 Console Application: Sum of digits	49
5.4 Console Application: Sum elements of an array	50
5.5 Windows Forms: Dealing with boxes	50
5.6 Windows Forms: Working with Strings	52
5.7 Windows Forms: Interacting with databases	54
5.8 Windows Forms: Interacting with XML files	58
Bibliography	62

Acronyms

C# – C Sharp

GUI – Graphical User Interface

LINQ – Language Integrated

Query SQL – Structured Query

Language VAT - Value Added

Tax

XML – Extensible Markup Language

Glossary

Array – a data structure that contains a group of elements. Typically these elements are all of the same data type, such as an integer or string. Arrays are commonly used in computer programs to organize data so that a related set of values can be easily sorted or searched.

Console – the combination of display monitor and keyboard (or other device that allows input). Another term for console is terminal.

LINQ – component of Microsoft .NET that adds query functionality in some .NET programming languages.

List – abstract data structure that implements an ordered collection of values, where the same value can occur more than once.

Prime number – natural numbers that have only two different divisors: the 1 and itself.

Relational database – a collective set of multiple data sets organized into tables, records and columns. RDBs establish a well-defined relationship between database tables. Tables communicate and share information, which facilitates data searchability, organization and reporting.

SQL Injection – SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands.

SQL Server – a SQL-based relational database management system designed for use in corporate applications, both on premises and in the cloud.

String – sequence of characters, generally used to represent words or phrases.

1. Introduction

1.1 Contextualization

It is agreed that different programming languages are suitable for the development of different applications. In fact, today, there is a range of languages suitable for the development of each type of application, be it a scientific application, a web application, an application for data management, etc. There is no single programming language suitable for any application development. On the contrary, it is common for the development of the same application to make use of different languages.

The evolution of programming languages leads to the emergence of new, more efficient languages, giving programming new horizons in the face of constant demands on the part of companies. Visual C#.NET is the programming language created by Microsoft and specially designed for the development of applications on the .NET platform. This language and its associated platform promise to radically change the way applications are developed for Windows, and also for the Internet.

Visual C# brings some important benefits like:

- A simple but simultaneously robust object-oriented programming language;
- Component oriented;
- Interoperability, scalability and performance;
- Rich library, particularly for building graphical applications.

1.2 Objectives

Having as a philosophy the learning by doing, this book proposes a step-by-step learning approach in which the concepts are accompanied by practical examples. It intends to give to the reader a general perception and knowledge of each component of Visual C#.NET and to solve practical exercises to consolidate the presented concepts. Finally, it presents a set of mini projects developed in Visual C#.NET that can help the reader to consolidate the addressed contents.

1.3 Book Structure

The book is organized in five chapters, respectively:

- Introduction – a brief contextualization of the book, objectives and its structure is given;
- The basic of console applications – provides a global overview about the process of creating console applications using Visual Studio;
- The basic of windows forms – provides a global overview about the process of creating windows forms applications using Visual Studio;
- C# syntax – gives a detailed overview about the syntax of C#. Several examples are provided associated to each component and C# instruction;
- Scenarios – numerous scenarios of creating mini projects in Visual C# are given. These scenarios target console applications and windows forms.

2. The basic of console applications

Console Applications are traditional applications without graphical interface. This type of application runs on command line with input and output information. Because information is written to and read from the console window, this type of application becomes a great way to learn programming techniques without having to worry about developing a graphical interface. To create a new console application in Visual Studio the following steps should be executed:

1. Start a new instance of Visual Studio;
2. On the menu bar, choose File -> New Project and then choose "Console Application" (Figure 1);

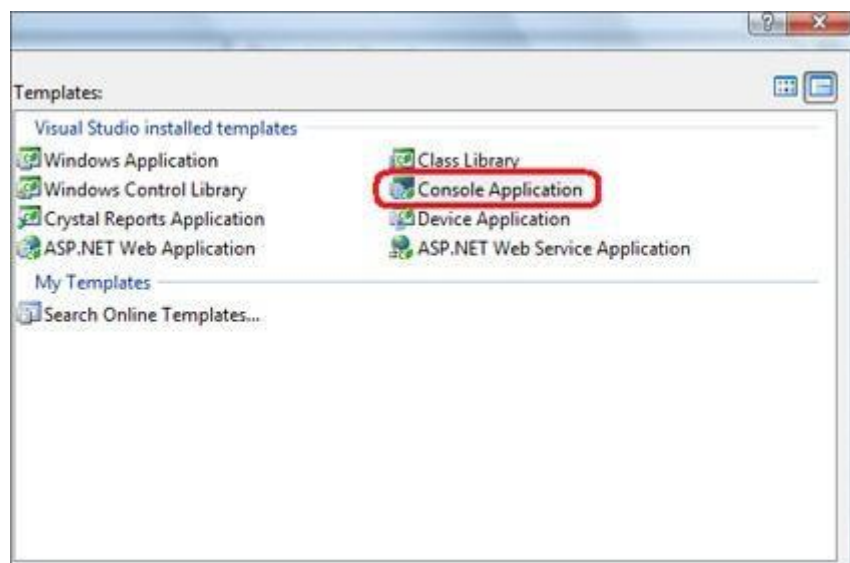


Figure 1 – Choose a Console Application

3. The name of the project must be specified in the name box. It is also important to select the folder in which the project will be saved;
4. The new created project appears in Solution Explorer. Associated to the project we have the "program.cs" file in which the code of the program will be developed.

After followed these steps the developer can start to write the code as it is depicted in Figure 2.



Figure 2 – Writing code in “program.cs”

Perhaps the easiest program that we can develop is the classic “Hello World”.

```
using System;
namespace
HelloWorld
{
    class Hello
    {
        static void Main()
        {
            Console.WriteLine("Hello World!");
            // Keep the console window open in debug
            mode. Console.WriteLine("Press any key to
            exit."); Console.ReadKey();
        }
    }
}
```

It is relevant to emphasize that comments in Visual C# can be placed in the code using two approaches:

- “//” – Inline comments;
- “/* and */” – comments among multiple

lines. Some examples are provided below.

```
// Line 1
// Line 2
```



```
// Line 3
/*
First Line to Comment
Second Line to
Comment Third Line to
Comment
```

A typical useful situation in a console environment application is to receive arguments from the command line. This situation is depicted in the example below, where we calculate the total number of arguments passed in the console.

```
class TestClass
{
    static void Main(string[] args)
    {
        // Display the number of command line
        arguments:
        System.Console.WriteLine(args.Length);
    }
}
```

Another more advanced example is given below. Args[] is considered an array of strings. The first thing to do is to check if the console received arguments. If it is positive, therefore the args is different of null. In the console all arguments are written. For that, a for loop is adopted for this purpose.

```
using
System;
class
Program
{
    static void Main(string[] args)
    {
        if (args == null)
        {
            Console.WriteLine("args is null");
        }
        else
        {
            for (int i = 0; i < args.Length; i++)
            {
                Console.WriteLine(args[i]);
            }
        }
    }
}
```

```
for (int i = 0; i < args.Length; i++)
{
    string argument = args[i];
    Console.Write("args index ");
    Console.Write(i); // Write index
    Console.Write(" is [");
    Console.Write(argument); // Write
    string Console.WriteLine("]");
}
}
Console.ReadLine();
}
}
```

3. The basic of windows forms

Windows Forms uses a set of managed libraries in .NET framework to design and develop graphical interface applications. It offers a graphical API to display data and manage user interactions with easier deployment and better security in client applications. It is built with event-driven architecture similar to Windows clients and hence, its applications wait for user input for its execution.

Windows Forms offers a designer tool in Visual Studio to use and insert controls in a form and range them as per desired layout, with provision for adding code to handle their events, which implement user interactions. Every control in Windows Forms application is a concrete instance of a class. The layout of a control in the Graphical User Interface (GUI) and its behavior are managed using methods and properties. Numerous controls are available such as text boxes, buttons, fonts, icons, combo boxes, area elements, and other graphic objects.

Some best practices for building Windows Forms applications are recommended below (Techopedia, 2017):

- Windows Forms classes can be extended, using inheritance, to design an application framework that can provide high level of abstraction and code reusability;
- Forms should be compact, with controls on it limited to a size that can offer minimum functionality. Additionally, the creation and removal of controls dynamically can reduce the number of static controls;
- Forms can be broken into chunks packaged in assemblies that can automatically update itself and can be easily managed with minimal effort;
- Designing the application to be stateless provides scalability and flexibility with ease for debugging and maintenance;
- Windows Forms applications should be designed based on the level of trust required, the need to request for permissions, and handle security exceptions wherever necessary;
- Windows Form cannot be passed across application domain boundary since they are not designed to be marshaled across application domains.

The first step to create a new Windows Form application is to initiate a new instance of Visual Studio and choose to create this type of project. This situation is illustrated in Figure 3. The user must provide the following information:

- Choose Visual C# as the programming language;
- In the project template choose the option “Windows Forms Application”;
- Choose the project name, solution name and the folder where solution will be saved.

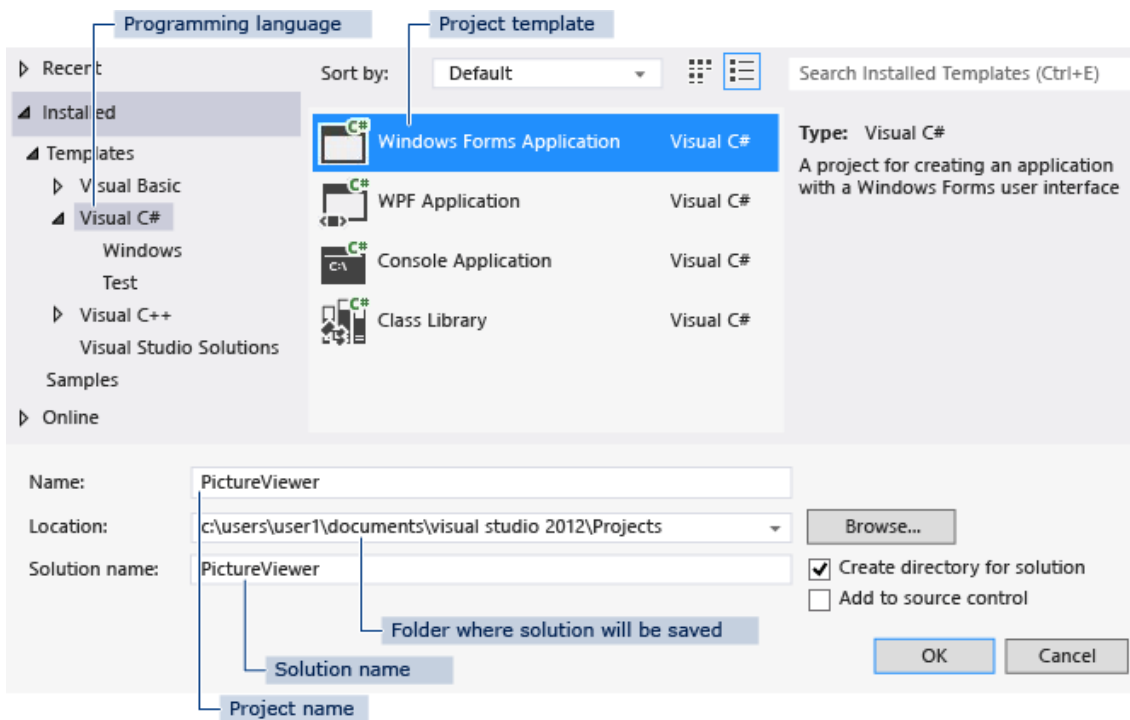


Figure 3 – Creation of a new Windows Form application

After this initial step the user can draw the graphical interface and associate the code to each object. The typical layout is given in Figure 4.

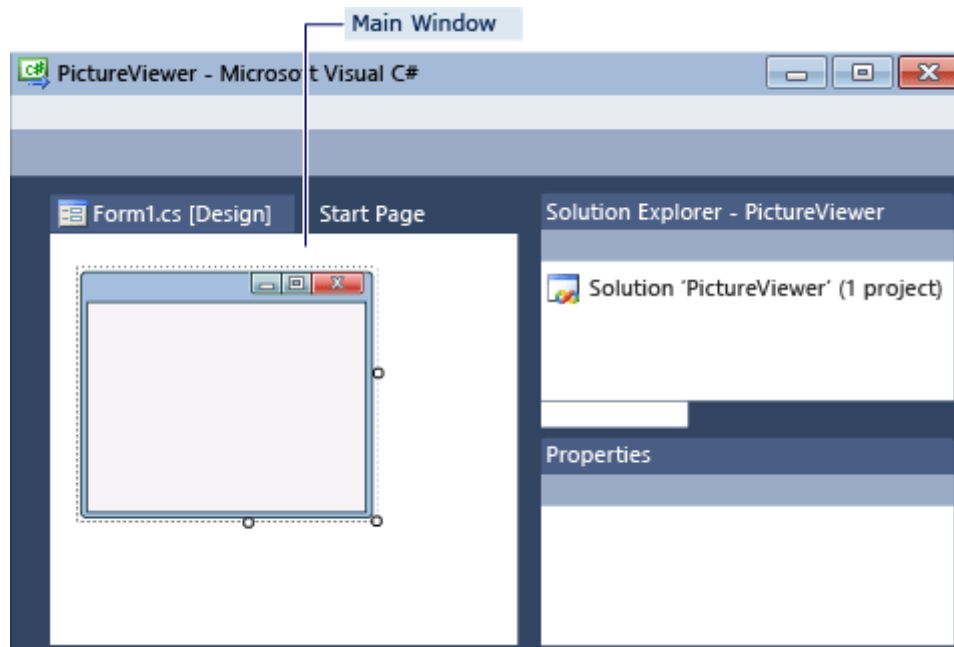


Figure 4 – Layout of a Windows Form application

The toolbox window can be used to draw objects in a Windows Form Application. Most common graphical objects are buttons, checkboxes, comboboxes, labels, monthcalendars and textboxes. Each object has properties, which can be easily accessed using the mouse

keyboard. Figure 5 gives an example of a property window, in which we can change several elements, such as the font, image, text, coordinates, etc.

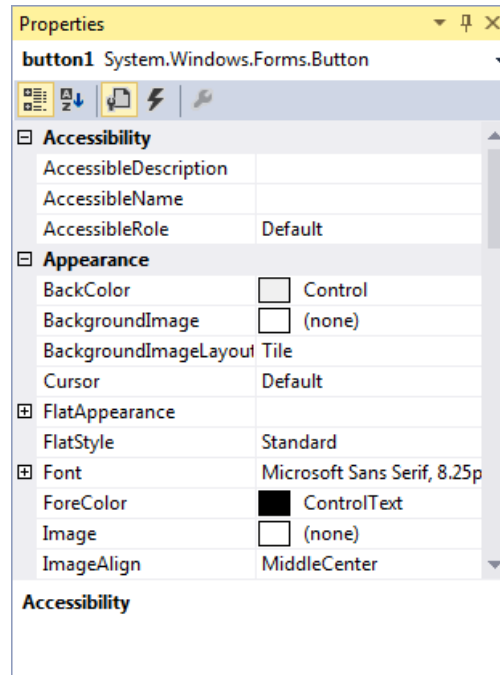


Figure 5 – Properties of an object

Like in the console application, here the most basic windows form application is also the traditional “hello world” program. For that, we will create a basic button and associate a message box when we press it.

```
using System;
using
System.ComponentModel;
using System.Drawing;
using System.Windows.Forms;

namespace FormWithButton
{
    public class Form1 : Form
    {
        public Button
        button1; public
        Form1()
        {
            button1 = new Button();
```

```
        button1.Location = new Point(30,
        30); button1.Text = "Click me";
        this.Controls.Add(button1);
        button1.Click += new EventHandler(button1_Click);
    }
    private void button1_Click(object sender, EventArgs e)
    {
        MessageBox.Show("Hello World");
    }
    [STAThread]
    static void
    Main()
    {
        Application.EnableVisualStyle
        s(); Application.Run(new
        Form1());
    }
}
```

4. C# syntax

4.1 Variables

Variables are stored in memory and allow keeping data during the program's execution. The value of each variable can be changed by the user directly or indirectly through a calculation process. There are several types of C # variables, some of which are quite complex, so attention is given at this stage to the most common and basic variable types (Figure 6).

Variable Type	Example
Decimal types	decimal
Boolean types	True or false value, as assigned
Integral types	int, char, byte, short, long
Floating point types	float and double
Nullable types	Nullable data types

Figure 6 – Basic types of variables

It is also important to look to the size and limits of each variable. This information is provided in Figure 7.

Data Types	Memory Size	Range
char	1 byte	-128 to 127
signed char	1 byte	-128 to 127
unsigned char	1 byte	0 to 127
short	2 byte	-32,768 to 32,767
signed short	2 byte	-32,768 to 32,767
unsigned short	2 byte	0 to 65,535
int	4 byte	-2,147,483,648 to -2,147,483,647
signed int	4 byte	-2,147,483,648 to -2,147,483,647
unsigned int	4 byte	0 to 4,294,967,295
long	8 byte	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
signed long	8 byte	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long	8 byte	0 - 18,446,744,073,709,551,615
float	4 byte	1.5×10^{-45} - 3.4×10^{38} , 7-digit precision
double	8 byte	5.0×10^{-324} - 1.7×10^{308} , 15-digit precision
decimal	16 byte	at least -7.9×10^{28} - 7.9×10^{28} , with at least 28-digit precision

Figure 7 – Size and limits of variables

The example below provides a small example performing initialization and operations with variables.

```
using System;
namespace
VariableDefinition { class
Program {
    static void Main(string[]
        args) { short a;
            int b ;
            double
            c;
            /* actual initialization
            */ a = 10;
            b = 20;
            c = a + b;
            Console.WriteLine("a = {0}, b = {1}, c = {2}", a,
            b, c); Console.ReadLine();
        }
    }
}
```

Typically it is common the need to convert one type of data to another type. There are two ways to do it:

- Implicit type conversion – these conversions are performed by C# in a type-safe manner. It typically involves conversions from smaller to larger integral types and conversions from derived classes to base classes;
- Explicit type conversion – these conversions are done explicitly by users using pre- defined functions. Explicit conversions require a cast operator.

In the example below we have a cast conversion from double to int.

```
using System;
namespace
TypeConversionApplication { class
ExplicitConversion {
    static void Main(string[]
        args) { double d =
            5673.74;
            int i;

            // cast double to
            int. i = (int)d;
            Console.WriteLine(
            i);
            Console.ReadKey(
            );
        }
    }
}
```

C# also provides built-in type conversion methods. Some of them include:

- ToDateTime – converts a type (integer or string type) to date-time structures;
- ToDecimal – converts a floating point or integer type to a decimal type;

- ToDouble – converts a number to a double type;
- ToInt32 – converts a type to a 32-bit integer;
- ToString – converts an element to a string.

In the example below we convert three different types to a string object.

```
using System;
namespace
TypeConversionApplication { class
StringConversion {
    static void Main(string[]
        args) { int i = 75;
        float f = 53.005f;
        double d =
        2345.7652;
        Console.WriteLine(i.ToString
        ());
        Console.WriteLine(f.ToString
        ());
        Console.WriteLine(d.ToStrin
        g()); Console.ReadKey();
```

4.2 Operators

Operators are use to process and calculate values. They can also be used for other purposes like testing conditions or for assignment processes. In general we have the following classes of operators:

- Arithmetic operators;
- Relational operators;
- Logical operators;
- Bitwise operators;
- Assignment operators;
- Unary operators;
- Ternary operators;
- Misc operators.

Figure 8 provides a concise overview of the operators that we may found in Visual C#.

	Operator	Type
Binary Operator	+, -, *, /, %	Arithmetic Operators
	<, <=, >, >=, ==, !=	Relational Operators
	&&, , !	Logical Operators
	&, , <<, >>, ~, ^	Bitwise Operators
	=, +=, -=, *=, /=, %=	Assignment Operators
Unary Operator	→ ++, --	Unary Operator
Ternary Operator	→ ? :	Ternary or Conditional Operator

Figure 8 – Overview of operators

Finally, it is also important to look to the precedence of operators. The associativity specifies if the operator will be evaluated from left to right or right to left. Most operators are evaluated from left to right, but there are some exceptions like the equality, ternary or assignment operators.

A simple example of using logical operators is given below. The first test condition returns True and the second returns False.

```
using System;
namespace
Operator
{
    class LogicalOperator
    {
        public static void Main(string[] args)
        {
            bool result;
            int firstNumber = 10, secondNumber = 20;

            // OR operator
            result = (firstNumber == secondNumber) || (firstNumber
            > 5); Console.WriteLine(result);

            // AND operator
            result = (firstNumber == secondNumber) && (firstNumber
            > 5); Console.WriteLine(result);
        }
    }
}
```

An example of using the ternary operator is given below. Basically, the ternary operator can be used to avoid the use of if-else statement. If the name is equal to Sam, then it returns "1", else it returns "-1".

```
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine(GetValue("Sam"));
        Console.WriteLine(GetValue("Tom"));
    }
    static int GetValue(string name)
    {
        return name == "Sam" ? 1 : -1;
    }
}
```

4.3 if-else statement

If and if-else statements allow the conditional execution of other commands. In the complete form, if-else, the -if command is executed when the condition is true, otherwise the -else command is executed. In the example below the input number is equal to 11. Therefore, in console, it will be written the following message: "It is odd number".

```
using System;

public static void Main(string[] args)
{
    int num = 11;
    if (num % 2 == 0)
    {
        Console.WriteLine("It is even number");
    }
    else
    {
        Console.WriteLine("It is odd number");
    }
}
```

"Else-if" statement can also be used together like in the below example.

```
static void Main(string[] args)
{
    int i = 10, j = 20;

    if (i > j)
    {
        Console.WriteLine("i is greater than j");
    }
}
```

```
        else if (i < j)
        {
            Console.WriteLine("i is less than j");
        }
        else
        {
            Console.WriteLine("i is equal to j");
        }
    }
}
```

4.4 Switch statement

The switch statement is an alternative of the “if-else” statement when we have multiple conditions. In the following example the “switch” is used to indicate the number passed by the user in the console. Readline() method is used to read the number and after that a new integer variable is created. The “convert” method is used to convert from string to int. Then, the num is tested in the switch statement. “Break” is used to exit from the switch statement.

```
public static void Main(string[] args)
{
    Console.WriteLine("Enter a number:");
    int num = Convert.ToInt32(Console.ReadLine());

    switch (num)
    {
        case 10: Console.WriteLine("It is 10"); break;
        case 20: Console.WriteLine("It is 20"); break;
        case 30: Console.WriteLine("It is 30"); break;
        default: Console.WriteLine("Not 10, 20 or 30"); break;
    }
}
```

4.5 For loop

For loop is used to iterate a part of a program several times. It is highly recommends when the number of iteration is fixed. Typically we have the following elements in a for loop: (i) initialization of a variable; (ii) check condition; and (iii) increment/decrement value. A simple example is given below.

```
using System;
public class ForExample
{
    public static void Main(string[] args)
    {
        for(int
            i=1;i<=10;i++){
```

```
}  
}  
}
```

It is possible to create a for loop without limits, like it is shown in the example below.

```
using System;  
namespace  
Loops {  
    class Program {  
        static void Main(string[]  
            args) { for (; ; ) {  
            Console.WriteLine("Hey! I am Trapped");  
        }  
    }  
}
```

4.6 While loop

The while loop is an alternative of the for loop. Typically the same functionality can be performed using both loops. However, if the number of iteration is not fixed, it is recommended to use the while loop. A simple example is given below. In this case, the for loop would give better performance.

```
Using System;  
public class WhileExample  
{  
    public static void Main(string[] args)  
    {  
        int i=1;  
        while(i<=10  
        )  
        {  
            Console.WriteLine(  
                i); i++;  
        }  
    }  
}
```

We can use the break instruction to break loop or switch statement. An example is given below. The numbers are written from 0 to 10. The break is used to exit from the while loop when i is higher than 10.

```
int i = 0;  
while  
(true)
```

```
Console.WriteLine("Value of i:  
{0}", i); i++;  
if (i >  
    10)  
    break
```

4.7 Arrays

Arrays are objects that have the same type and have contiguous memory location. In Visual C# the array index starts from zero. Arrays offer the following features:

- Code optimization;
- Random access;
- Easy to manipulate, traverse and sort data.

A visual example of an array structure is given in Figure 9.

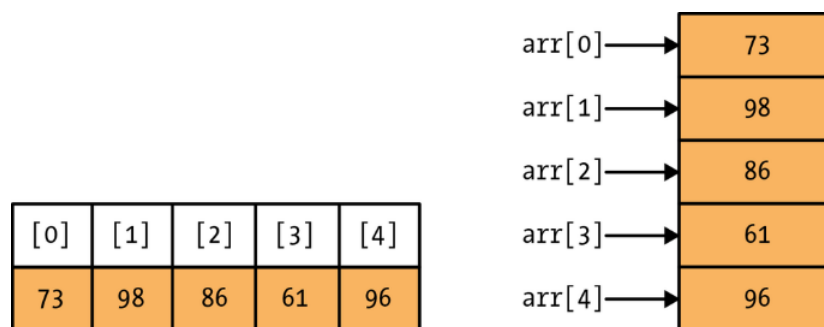


Figure 9 – Structure of an array (David Grossman, 2013)

Visual C# offers three types of arrays:

- Single dimensional array;
- Multidimensional array;
- Jagged array.

Single dimensional array is the most common and easiest way to have an array. It can be created by declaring:

- `Int[] arr = new int[5];`

This declaration means that we have a single dimensional array composed of five elements. A more complete example is given below. In this example we have generally three phases: (i) creating array; (ii) initializing array; and (iii) traversing array. Position 1 and 3 of the array are not initialized. A for loop is used to traversing the array.

```
using System;  
public class ArrayExample
```

```
{
    public static void Main(string[] args)
    {
        int[] arr = new int[5]; //creating
        array arr[0] = 10; //initializing array
        arr[2] = 20;
        arr[4] = 30;

        //traversing array
        for (int i = 0; i < arr.Length; i++)
        {
            Console.WriteLine(arr[i]);
        }
    }
}
```

Foreach loop can also be used to traversing the elements of an array. An example is given below.

```
using System;
public class ArrayExample
{
    public static void Main(string[] args)
    {
        int[] arr = { 10, 20, 30, 40, 50 }; //creating and initializing array

        //traversing array
        foreach (int i in
            arr)
        {
            Console.WriteLine(i);
        }
    }
}
```

We can also have multidimensional arrays that basically have more than one dimension. An example is given below. In the beginning of the process the array is initialized. Then, each element is written in the console using two for loops.

```
using System;
namespace ArrayApplication
{
    static void Main(string[]
        args) {
```

```
int[,] a = new int[5, 2] {{0,0}, {1,2}, {2,4}, {3,6}, {4,8} };
int i, j;
/* output each array element's value
*/ for (i = 0; i < 5; i++) {
    for (j = 0; j < 2; j++) {
        Console.WriteLine("a[{0},{1}] = {2}", i, j,
            a[i,j]);
    }
}
Console.ReadKey();
}
```

Finally, an example of a jagged array is given. A Jagged array is basically an array of arrays. It can be used to store more efficiently many rows of varying lengths. Any type of data, reference or value, can be used.

```
using
System;
class
Program
{
    static void Main()
    {
        // Declare local jagged array with 3
        rows. int[][] jagged = new int[3][];
        // Create a new array in the jagged array, and
        assign it. jagged[0] = new int[2];
        jagged[0][0] = 1;
        jagged[0][1] = 2;
        // Set second row, initialized to
        zero. jagged[1] = new int[1];
        // Set third row, using array
        initializer. jagged[2] = new int[3] {
        3, 4, 5 };
        // Print out all elements in the jagged
        array. for (int i = 0; i < jagged.Length;
        i++)
```



```
{  
    Console.Write(innerArray[a] + " ");  
}  
Console.WriteLine();  
}  
}  
}
```

4.8 Arrays class

Array class is responsible to provide methods and operations to deal with arrays. These methods turn easier the process of creating, manipulating, searching and sorting elements of an array.

The most common proprieties offered by an array class are:

- **IsFixedSize** – it is used to get a value indicating whether the array has a fixed size or not;
- **Length** – it is used to get the total number of elements in all the dimensions of the array;
- **Rank** – it is used to get the number of dimensions of the array.

Looking to the methods, the most useful are:

- **Clone()** – it is used to create a shallow copy of the array;
- **CopyTo(Array, Int32)** – it copies all the elements of an array to another array starting at the specific destination array index;
- **IndexOf(Array, Object)** – it is used to search for a specific object and returns the index of its first occurrence in an array;
- **Reverse(Array)** – it is used to reverse the sequence of the elements in the entire array;
- **Sort(Array)** – it is used to sort the elements in an entire array.

Below we may find an example of using several operation of an array class. The results are written in the console using the `PrintArray()` method.

```
using System;  
namespace CSharpProgram  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            // Creating an array  
            int[] arr = new int[6] { 5, 8, 9, 25, 0, 7 };  
            // Creating an empty  
            array int[] arr2 = new  
            int[6];  
            // Displaying length of array
```

```
Console.WriteLine("length of first array: "+arr.Length);
// Sorting
array
Array.Sort(arr
);
Console.Write("First array elements: ");
// Displaying sorted
array PrintArray(arr);
// Finding index of an array element
Console.WriteLine("\nIndex position of 25 is "+Array.IndexOf(arr,25));
// Coping first array to empty array
Array.Copy(arr, arr2, arr.Length);
Console.Write("Second array elements: ");
// Displaying second
array PrintArray(arr2);
Array.Reverse(arr);
Console.Write("\nFirst Array elements in reverse
order: "); PrintArray(arr);
}
// User defined method for iterating array
elements static void PrintArray(int[] arr)
{
    foreach (Object elem in arr)
    {
        Console.Write(elem+" ");
    }
}
}
```

4.9 Functions

A function is used to execute statements within a specific block. It has fundamentally two purposes: (i) increase the readability of the code; and (ii) help in reusing code. When declaring a new function, three kinds of elements must be defined:

- Function name – it is unique name that is used to call the function;
- Return type – is used to specify the data type of function return value;
- Parameters – list of arguments that we can pass to the function during call.

In the first example below we defined a function that uses a string parameter and returns also a string. Two strings are written in the console: (i) one inside the show() function; and (ii) another inside the main() function.

```
using System;
namespace FunctionExample
{
    class Program
    {
        // User defined function
        public string Show(string message)
        {
```

```
        Console.WriteLine("Inside Show  
Function"); return message;  
    }  
    // Main function, execution entry point of the  
    program static void Main(string[] args)  
    {  
        Program program = new Program();  
        string message = program.Show("My  
Name"); Console.WriteLine("Hello  
"+message);  
    }  
}
```

A function can also receive “out” parameters. It works like a reference-type, except that it does not require variable to initialize before passing. An example is given below. The values before calling the show() function are equal to 50 and 100; after executing the function, the new variables are equal to 25 for both variables.

```
using System;  
namespace OutParameter  
{  
    class Program  
    {  
        // User defined function  
        public void Show(out int a, out int b) // Out parameter  
        {  
            int square =  
                5; a =  
                square;  
            b = square;  
            // Manipulating value  
            a *= a;  
            b *= b;  
        }  
        // Main function, execution entry point of the  
        program static void Main(string[] args)  
        {  
            int val1 = 50, val2 = 100;  
            Program program = new Program(); // Creating Object  
            Console.WriteLine("Value before passing \n val1 = " + val1+" \n val2 =  
"+val2); program.Show(out val1, out val2); // Passing out argument
```

```
        Console.WriteLine("Value after passing \n val1 = " + val1 + " \n val2 = " + val2);  
    }  
}  
}
```

4.10 Class

A class is the grouping of objects with the same data structure defined by attributes or properties and operations. In short, classes are descriptions of objects. An example of using a class only with attributes is provided below. We create a new Student class with two attributes: (i) id; and (ii) name. Both attributes are written in the console.

```
using System;  
public class Student  
{  
    int id;  
    String name;  
    public static void Main(string[] args)  
    {  
        Student s1 = new Student();//creating an object of  
        Student s1.id = 101;  
        s1.name = "My name";  
        Console.WriteLine(s1.id);  
        Console.WriteLine(s1.name);  
    }  
}
```

A more complete example is given below where we have an example with a class that also offers two methods. The insert() method is used to add a new id and name. The display() method is used to write the name of the student in the console. Two new students' objects are created in the main function.

```
using System;  
public class Student  
{  
    public int id;  
    public String name;  
    public void insert(int i, String n)
```

```
{
    id = i;
    name =
    n;
}
public void display()
{
    Console.WriteLine(id + " " + name);
}
}
class TestStudent{
    public static void Main(string[] args)
    {
        Student s1 = new
        Student(); Student s2 =
        new Student();
        s1.insert(101, "Peter");
        s2.insert(102,
        "Tom");
        s1.display();
        s2.display();
    }
}
```

The “this” syntax can be used to refer to the current instance of the class. This approach can be used to refer current class instance variable, to pass current object as a parameter to another method and to declare indexers. A simple example how to adopt this approach is given below. In this case we create a new Employee() class that contains information on id, name, and salary.

```
using System;
public class Employee
{
    public int id;
    public String
    name; public float
    salary;
    public Employee(int id, String name, float salary)
    {
    }
```

```
        this.name =  
        name; this.salary  
        = salary;  
    }  
    public void display()  
    {  
        Console.WriteLine(id + " " + name+" "+salary);  
    }  
}  
class TestEmployee{  
    public static void Main(string[] args)  
    {  
        Employee e1 = new Employee(101, "Peter",  
        50000f); Employee e2 = new Employee(102,  
        "Tom", 32000f); e1.display();  
        e2.display();  
    }  
}
```

It is possible to have static classes, which are normal classes but they can't be instantiated. Additionally, it can have only static members. Static classes contain only static members, cannot be instantiated, and cannot contain instance constructors. An example is given below. There is a static attribute called "PI" and a static method entitled "cube" that receives an integer parameter.

```
using System;  
public static class MyMath  
{  
    public static float PI=3.14f;  
    public static int cube(int n){return n*n*n;}  
}  
class TestMyMath{  
    public static void Main(string[] args)  
    {  
        Console.WriteLine("Value of PI is: "+MyMath.PI);  
        Console.WriteLine("Cube of 3 is: " +  
        MyMath.cube(3));  
    }  
}
```

```
}
```

A class offer inheritance that is a process in which one object acquires all the properties of its parent object automatically. The class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The biggest advantage of inheritance is to improve the code reusability. An example is given below. A programmer is a particular case of an employee. It has access to the generic salary of the employee and it has access to the bonus that is specific of the Programmer class. The information regarding salary and bonus is written in the console.

```
using System;
public class Employee
{
    public float salary = 40000;
}
public class Programmer: Employee
{
    public float bonus = 10000;
}
class TestInheritance{
    public static void Main(string[] args)
    {
        Programmer p1 = new Programmer();
        Console.WriteLine("Salary: " +
            p1.salary); Console.WriteLine("Bonus:
            " + p1.bonus);
    }
}
```

4.11 Structs

A struct is a special variable that contains several other variables internally generally of different types. The internal variables contained by the struct are called members of the struct. The purpose of a struct is to allow, when storing the data of the same entity, this can be done with a single variable. In the example below we create a new rectangle structure. A rectangle is composed of width and height. The area of rectangle is calculated multiplying the width with height.

```
using System;
public struct Rectangle
```

```
{  
    public int width, height;  
  
}  
public class TestStructs  
{  
    public static void Main()  
    {  
        Rectangle r = new  
        Rectangle(); r.width = 4;  
        r.height = 5;  
        Console.WriteLine("Area of Rectangle is: " + (r.width * r.height));  
    }  
}
```

Another example of using structs is given below. In this case we create a new struct for a book containing information on book_id, title, author and subject. Two books are created, specified and printed.

```
using  
System;  
struct Books  
{  
    public string title;  
    public string  
    author; public  
    string subject;  
    public int book_id;  
};  
public class testStructure {  
    public static void Main(string[] args) {  
        Books Book1; /* Declare Book1 of type Book */  
        Books Book2; /* Declare Book2 of type Book */  
        /* book 1 specification */  
        Book1.title = "C  
        Programming";  
        Book1.author = "Nuha Ali";
```



```
Book2.title = "Telecom Billing";
Book2.author = "Zara Ali";
Book2.subject = "Telecom Billing Tutorial";
Book2.book_id = 6495700;
/* print Book1 info */
Console.WriteLine( "Book 1 title : {0}", Book1.title);
Console.WriteLine("Book 1 author : {0}",
Book1.author);
Console.WriteLine("Book 1 subject : {0}", Book1.subject);
Console.WriteLine("Book 1 book_id :{0}", Book1.book_id);
/* print Book2 info */
Console.WriteLine("Book 2 title : {0}", Book2.title);
Console.WriteLine("Book 2 author : {0}", Book2.author);
Console.WriteLine("Book 2 subject : {0}", Book2.subject);
Console.WriteLine("Book 2 book_id : {0}",
Book2.book_id); Console.ReadKey();
}
}
```

4.12 Strings

In programming a string is a sequence of characters generally used to represent words, phrases or texts in a program. The example below shows two ways to represent a string object. "S1" is declared initially as a string and "ch" is declared as an array of char objects. Then, we create a new string "s2" that is composed of "ch".

```
using System;
public class StringExample
{
    public static void Main(string[] args)
    {
        string s1 = "hello";
        char[] ch = { 'c', 's', 'h', 'a', 'r', 'p'
    }; string s2 = new string(ch);
        Console.WriteLine(s1);
        Console.WriteLine(s2);
    }
}
```

```
}
```

The string class offers some useful methods to perform the following operations:

- Compare (string,string) – it is used to compare two specific String objects. It returns an integer that indicates their relative position in the sort order;
- CompareTo (String) – it is used to compare this string instance with a specific String object. It indicates whether this instance precedes, follows, or appears in the same position in the sort order as the specific string;
- Concat (String, String) – it is used to concatenate two specified instances of string;
- Contains (String) – it is used to return a value indicating whether a specific substring occurs within this string;
- Copy (string) – it is used to create a new instance of String with the same value as a specified String;
- Equals (String, String) – it is used to determine that two specified String objects have the same value;
- GetHashCode() – it returns the hash code for this string;
- IndexOf (String) – it is used to report the zero-based index of the first occurrence of the specified string in this instance;
- Insert (Int, String) – it is used to return a new string in which a specific string is inserted at a specified index position;
- Remove (Int) – it is used to return a new string in which all the characters in the current instance, beginning at a specified position and continuing through the last position, have been deleted;
- Split (Char[]) – it is used to split a string into substrings that are based on the characters in an array;
- Substring (Int) – it is used to retrieve a substring from this instance. The substring starts at a specified character position and continues to the end of the string;
- ToLower() – it is used to convert a string into lowercase;
- ToUpper() – it is used to convert a string into uppercase;
- Trim() – it is used to remove all leading and trailing white-space characters from the current string object;

The above list is not exhaustive, but intends to give an overview about the main functions offered by String object.

An example of using the IndexOf() method is given below. Basically it returns a value different of -1 if the “dog” name is found in the string.

```
using  
System;  
class  
Program
```

```
{
    // The input string.
    const string value = "The dog is here.";
    // Test with IndexOf
    method. if
    (value.IndexOf("dog") != -
    1)
    {
        Console.WriteLine("string contains dog!");
    }
}
```

In the example below we use the split() method to divide a string in several words.

```
using
System;
class
Program
{
    static void Main()
    {
        string s = "there is a cat and a dog";
        // Split string on spaces.
        // ... This will separate all the
        words. string[] words = s.Split(' ');
        foreach (string word in words)
        {
            Console.WriteLine(word);
        }
    }
}
```

4.13 Exceptions

An exception is an indication that some type of exceptional condition occurred during the execution of the program. Therefore, exceptions are associated with error conditions that could not be verified during program compilation. The two activities associated with handling an exception are:

- Generation – signaling that an exceptional condition (for example, an error) has occurred;

- Capture – the handling (treatment) of the exceptional situation where the actions required for the recovery of the error situation are defined.

For each exception that can occur during code execution, a block of handling actions (an exception handler) must be specified. The structure of exceptions handling is given in Figure

10. The goal is each block is the following:

- Try – encloses the statements that might throw an exception;
- Catch – handles the exceptions thrown by the try block;
- Finally – optional block that is always executed after the execution of the exception.

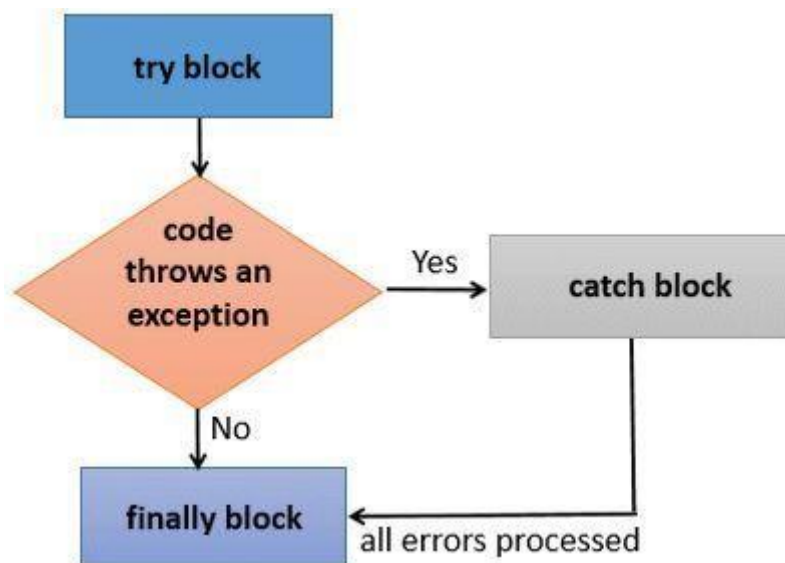


Figure 10 – Structure of exceptions handling (Arora, 2015)

Two categories of exceptions exist: (i) exceptions that are generated by the application; and (ii) those generated by the runtime. All exceptions the derived from System.Exception class like indicated in Figure 11.

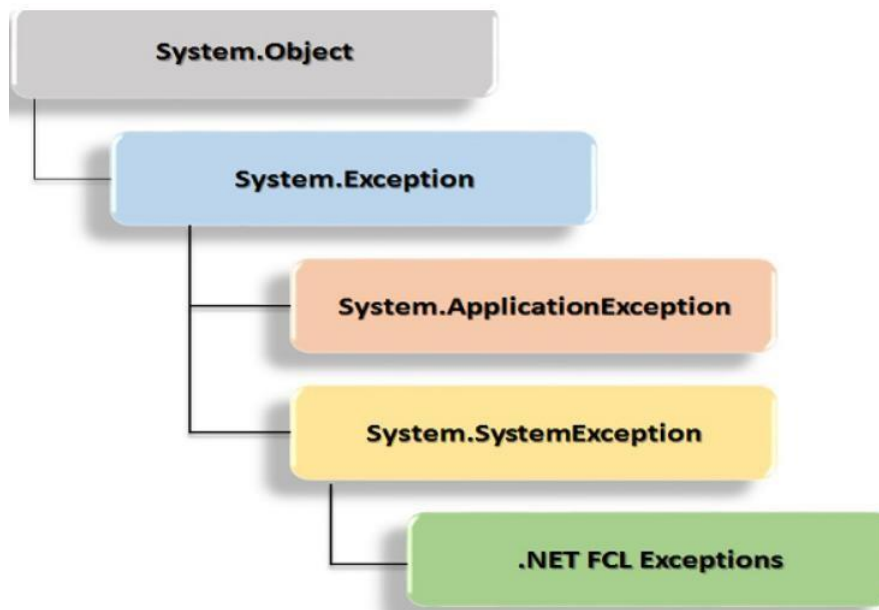


Figure 11 – Exception handling hierarchy (Kanjilal, 2015)

Common exception classes may be:

- **System.DivideByZeroException** – handles the error generated by dividing a number with zero;
- **System.NullReferenceException** – handles the error generated by referencing the null object;
- **System.InvalidCastException** – handles the error generated by invalid typecasting;
- **System.IO.IOException** – handles the input/output errors;
- **System.FieldAccessException** – handles the errors generated by invalid private or protected field access.

In the example below we provide a simple example how to use the try/catch statement. The application will launch an exception saying “System.DivideByZeroException: Attempted to divide by zero”.

```
using System;
public class ExExample
{
    public static void Main(string[] args)
    {
        try
        {
            int a = 10;
            int b = 0;
            int x = a /
            b;
        }
        catch (Exception e) { Console.WriteLine(e); }
        finally { Console.WriteLine("Finally block is
        executed"); } Console.WriteLine("Rest of the
```

```
}  
}
```

It is also possible to create user-defined exceptions. It is used to personalize exceptions according to programming needs. To do this, we need to inherit Exception class. In the example below a custom exception was created to guarantee that all ages are greater than 18.

```
using System;  
public class InvalidAgeException : Exception  
{  
    public InvalidAgeException(String message)  
        : base(message)  
    {  
    }  
}  
public class TestUserDefinedException  
{  
    static void validate(int age)  
    {  
        if (age < 18)  
        {  
            throw new InvalidAgeException("Sorry, Age must be greater than 18");  
        }  
    }  
    public static void Main(string[] args)  
    {  
        try  
        {  
            validate(12);  
        }  
        catch (InvalidAgeException e) {  
            Console.WriteLine(e); } Console.WriteLine("Rest  
of the code");  
    }  
}
```

4.14 Files

Working with files can be done in C# using the FileStream class. This class can be used to perform synchronous and asynchronous read and write operations. In the example below we used the ReadByte() function to write in the console the contents of a file. In the end the file is closed using the close() method.

```
using System;  
using  
System.IO;  
public class FileStreamExample  
{  
    public static void Main(string[] args)  
    {
```

```
int i = 0;
while ((i = f.ReadByte()) != -1)
{
    Console.Write((char)i);
}
f.Close();
}
```

Other alternative to read information from a file is to use the StreamReader class. It provides two methods to read data:

- Read() – read one character;
- ReadLine() – read a single line from the file.

An example of using the StreamReader class is given below. In this example all lines of the file are read and written in the console.

```
using System;
using
System.IO;
public class StreamReaderExample
{
    public static void Main(string[] args)
    {
        FileStream f = new FileStream("c:\\example_file.txt",
        FileMode.OpenOrCreate); StreamReader s = new StreamReader(f);

        string line = "";
        while ((line = s.ReadLine()) != null)
        {
            Console.WriteLine(line);
        }
        s.Close();
        f.Close();
    }
}
```

Now our intention is to write information into a file. For that, we can use the StreamWriter() class. Also in this situation two methods can be used: (i) write(); and writeln(). In the example below we write a single line of data into the file.

```
using System;
using
System.IO;
public class StreamWriterExample
{
    public static void Main(string[] args)
```

```
FileStream f = new FileStream("c:\\output.txt",  
    FileMode.Create); StreamWriter s = new StreamWriter(f);  
s.WriteLine("hello visual  
c#"); s.Close();  
f.Close();  
Console.WriteLine("File created successfully.");  
}  
}
```

We can also use the `FileInfo` class to deal with file operations in C#. For instance, we can use it to create, delete and read files. `FileInfo` class also provides some useful properties and methods.

List of useful properties:

- `CreationTime` – it is used to get or set the creation time of the current file;
- `DirectoryName` – it is used to get a string representing the directory's full path;
- `Exists` – it is used to indicate whether a file exists;
- `LastAccessTime` – it is used to get or set the time from the current file;
- `Length` – it is used to get the size of the current file;
- `Name` – it is used to get the name of the file.

List of useful methods:

- `AppendText()` – it is used to create a `StreamWriter` that appends text to the file;
- `CopyTo (String)` – it is used to copy an existing file to a new file;
- `Delete()` – it is used to permanently delete a file;
- `MoveTo (String)` – it is used to move a specified file to a new location;
- `Open (FileMode)` – it is used to open a file in the specified mode.

In the example below we use `FileInfo` class to create a new file and add text to it.

```
using System;  
using  
System.IO;  
namespace CSharpProgram  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {
```



```
{
    // Specifying file
    location string loc =
        "C:\\test.txt";
    // Creating FileInfo instance
    FileInfo file = new
        FileInfo(loc);
    // Creating an file instance to
    write StreamWriter sw =
        file.CreateText();
    // Writing to the file
    sw.WriteLine("This text is written to the file by using StreamWriter
        class."); sw.Close();
} catch(IOException e)
{
    Console.WriteLine("Something went wrong: "+e);
}
}
```

4.15 Collections

Collections are generally complex data structures that can store objects. On the contrary of arrays, which have size limit, collections can grow or shrink dynamically. There are several types of collections, such as lists, stacks, queues, dictionaries or hashsets. In the example below we use a list to store elements. The add() method is used to insert new elements in the list.

```
using System;
using
System.Collections.Generic;
public class ListExample
{
    public static void Main(string[] args)
    {
        // Create a list of strings
        var names = new
            List<string>();
        names.Add("Tom");
    }
}
```

```
names.Add("Irfan");
// Iterate list element using foreach
loop foreach (var name in names)
{
    Console.WriteLine(name);
}
}
```

Other alternative is to use dictionaries that explore the concept of hashtables. It stores values on the basis of an unique key. It can be used to easily search or remove elements. In the example below we create a new dictionary and we associate three elements to it.

```
using System;
using
System.Collections.Generic;
public class
DictionaryExample
{
    public static void Main(string[] args)
    {
        Dictionary<string, string> names = new Dictionary<string,
string>(); names.Add("1","Tom");
names.Add("2","Peter");
names.Add("3","James");
foreach (KeyValuePair<string, string> kv in names)
{
    Console.WriteLine(kv.Key+" "+kv.Value);
}
}
```

4.16 Databases

A very useful functionality of C# is the establishment of physical communication with an external relational database. In this book, we will provide a brief overview how to do it with a SQL Server Database.

The first step is to define a new SqlConnection instance that takes a connection string as argument.

```
connetionString="Data Source=ServerName; Initial Catalog=DatabaseName;User ID=UserName;Password=Password"
```

In a named instance of SQL Server it is also necessary to specify the server location.

When the connection is established, SQL Commands will execute with the help of the Connection Object and retrieve or manipulate the data in the database. Once the Database activities are over, Connection should be closed and release the Data Source resources.

A complete example how to establish a connection to a SQL Server Database is given below.

```
using System;
using System.Windows.Forms;
using System.Data.SqlClient;
namespace
WindowsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void button1_Click(object sender,
            EventArgs e)
        {
            string connetionString = null;
            connetionStrin = "Dat Source=ServerName;Ini
            Catalog=DatabaseName;User
            ID=UserName;Password=Password"
            cnn = new
            SqlConnection(connetionString); try
            {
                cnn.Open();
                MessageBox.Show ("Connection Open
                ! "); cnn.Close();
            }
            catch (Exception ex)
            {
                MessageBox.Show("Cannot open connection ! ").
```

```
}  
}  
}  
}
```

Once connected to the database, we can execute the set of SQL commands.

```
SqlCommand command = new SqlCommand("SELECT * FROM TableName", conn);
```

Parameterizing the query is done by using the SqlParameter passed into the command. It is a good approach to avoid SQL Injection.

```
// Create the command
```

```
SqlCommand command = new SqlCommand("SELECT * FROM TableName WHERE  
FirstColumn  
= @0", conn);
```

```
// Add the parameters.
```

```
command.Parameters.Add(new
```

```
SqlParameter("0", 1));
```

4.17 XML

XML can be used in information systems to share data. In C# it becomes possible to create, read and edit content of an XML document. We start by presenting the following example of an XML file.

```
<?xml version="1.0" encoding="UTF-8"?>  
<peessoas>  
  <pessoa codigo="1" nome="Steve Jobs" telefone="2222-2222"/>  
  <pessoa codigo="2" nome="Bill Gates" telefone="3333-3333"/>  
  <pessoa codigo="1" nome="Steve Ballmer" telefone="4444-4444"/>  
</peessoas>
```

The first method lists persons in the database. All attributes of the "Pessoa" in the XML file are passed to the "Pessoa" class.

```
public static List<Pessoa> ListarPessoas()  
{  
  List<Pessoa> pessoas = new  
  List<Pessoa>(); XElement xml =  
  XElement.Load("Pessoas.xml");  
  foreach(XElement x in xml.Elements())
```

```
Pessoa p = new Pessoa()
{
    codigo =
        int.Parse(x.Attribute("codigo").Value),
    nome = x.Attribute("nome").Value,
    telefone = x.Attribute("telefone").Value
};
pessoas.Add(p);
}
return pessoas;
```

The next method is responsible for inserting new records and will receive as a parameter an object of type Person.

```
public static void AdicionarPessoa(Pessoa p)
{
    XElement x = new XElement("pessoa");
    x.Add(new XAttribute("codigo",
        p.codigo.ToString())); x.Add(new
    XAttribute("nome", p.nome));
    x.Add(new XAttribute("telefone",
        p.telefone)); XElement xml =
    XElement.Load("Pessoas.xml");
    xml.Add(x);
}
```

It is also possible to exclude elements in a XML file.

```
public static void ExcluirPessoa(int codigo)
{
    XElement xml =
        XElement.Load("Pessoas.xml");
    XElement x = xml.Elements().Where(
        e => e.Attribute("codigo").Value.Equals(codigo.ToString())).First();
    if (x != null)
    {
        x.Remove();
    }
}
```

```
}
```

Finally, the last piece of code can be used to edit an element in a XML file. The element is edited and saved on the same XML file.

```
public static void EditarPessoa(Pessoa
pessoa)
{
    XEleme      x      =      xml.Elements().Where      =
p.Attribute("codigo").Value.Equals(pessoa.codigo.ToString()).
First();
    if (x != null)
    {
        x.Attribute("nome").SetValue(pessoa.nome);
        x.Attribute("telefone").SetValue(pessoa.telefo
ne);
    }
    xml.Save("Pessoas.xml");
}
```

To create a new XML file the following code can be used. It creates a new XML file entitled "Studentx.xml" that contains a new XML file with information regarding his/her name, email and city.

```
XDocument document = new
    XDocument( new
        XDeclaration("0.1", "utf-8", "yes"),
        new XElement("Students",
            new XElement("Student",
                new XElement("Name", "Paul"),
                new
                    XElement("Email", "paul@email.com"),
                    new XElement("City", "Dallas")
            )
        )
    )
```

Other example is given below, but in this last situation we have an "id" attribute.

```
XDocument document1 = new
    XDocument( new
        XDeclaration("0.1", "utf-8", "yes"),
```

```
new
  XElement("Student",
    new
      XAttribute("Id", "1"),
    new XElement("Name", "Paul"),
    new XElement("Email",
      "paul@email.com"), new
      XElement("City", "Dallas")
    )
  )
```

5. Scenarios

5.1 Console Application: Price of products

In this first scenario we present a basic situation in which we intend to calculate the final price of a product. The final price is composed of three elements:

- Preço_base – the base price of the product without any tax;
- Desconto – percentage value of the discount;
- IVA – VAT tax.

All these elements are defined in the main function. Two elements are calculated and written in the console: (i) the value of the price with discount; and (ii) the final price of the product.

```
namespace Exercicio1
{
    class Program
    {
        static void Main(string[] args)
        {
            double
            total_desconto;
            double iva = 0.23;
            double preço_base =
            20; double desconto =
            0.10; double total;

            total_desconto = preço_base * (1 - 0.10);

            Console.WriteLine("Total com desconto: "+
            total_desconto); total = total_desconto * (1 + iva);

            Console.WriteLine("Total final: " + total);

            Console.ReadLine();
        }
    }
}
```

5.2 Console Application: Prime numbers

This scenario intends to test if a number is prime or not. The first thing to do is to receive a number from the console. For that we use the `Console.ReadLine()` method. After that a for loop is created to test if the number can be divided by one of its divisors. The remainder of the division is calculated using the “%” operator. On the end, if the flag is still equal to zero, then we can assume that the number is prime.

```
using System;
public class PrimeNumberExample
```



```
{
public static void Main(string[] args)
{
    int n, i, m=0, flag=0;
    Console.Write("Enter the Number to check
    Prime: "); n = int.Parse(Console.ReadLine());
    m=n/2;
    for(i = 2; i <= m; i++)
    {
        if(n % i == 0)
        {
            Console.Write("Number is not
            Prime."); flag=1;
            break;
        }
    }
    if (flag==0)
        Console.Write("Number is Prime.");
    }
}
```

5.3 Console Application: Sum of digits

The idea of this scenario is to calculate the sum of digits. The algorithm for performing it is composed of the following steps:

1. Get number by the user;
2. Get the remainder of the number;
3. Sum the remainder of the number;
4. Divide the number by 10;
5. Repeat the step 2 while the number is greater than 0.

```
using System;
public class SumExample
{
    public static void Main(string[] args)
    {
        int n,sum=0,m;
        Console.Write("Enter a number:
        "); n=
        int.Parse(Console.ReadLine());
        while(n>0)
        {
            m=n%10;
            sum=sum+
            m; n=n/10;
        }
        Console.Write("Sum is= "+sum);
    }
}
```

5.4 Console Application: Sum elements of an array

The idea of this program is to sum five elements of an array. The program offers a Sum() function that receives an array as argument. Using a for loop this function sums all elements of an array object. In the main function the program starts by requesting to the user to give five elements to be added to an array object. After that, the program calls the Sum() function and writes the sum on the console.

```
using System;

namespace Exercicio3f
{
    class Program
    {
        public static int Sum(int[] arr1)
        {
            int tot = 0;
            for (int i = 0; i < arr1.Length;
                i++) tot += arr1[i];
            return tot;
        }
        public static void Main()
        {
            int[] arr1 = new int[5];
            Console.WriteLine("Calcular soma dos elementos de um array");

            Console.WriteLine("Escreva 5 elementos para o
            array."); for (int j = 0; j < 5; j++)
            {
                Console.Write("elemento {0} : ", j);
                arr1[j] = Convert.ToInt32(Console.ReadLine());
            }
            Console.WriteLine("Soma: " + Sum(arr1));
            Console.ReadKey();
        }
    }
}
```

5.5 Windows Forms: Dealing with boxes

In windows forms it is fundamental to know how to write and read information from textboxes and comboboxes. Figure 12 provides an overview about the considered scenario. Four different visual elements are used:

- Combobox – to list all the customers;
- Buttons – three buttons are used: “initialize”, “add item”, and “remove”;
- Textbox – to place the name of the customer, which can be added;
- Label – to write the total number of items.

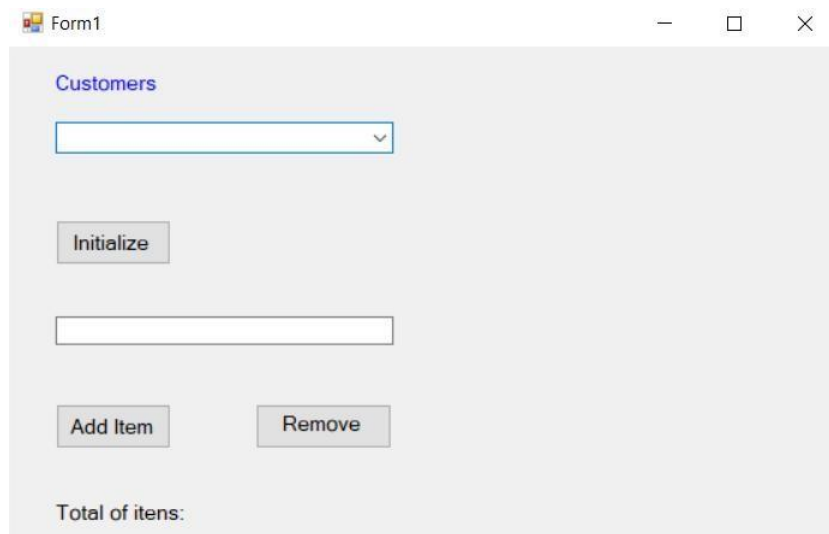


Figure 12 – Dealing with boxes scenario

The user should start to initialize the elements in the combobox. Three customers are added by pressing the “initialize” button. Then the user can add or remove items. A new name for the customer must be provided to add a new item. If the user wants to remove an item, then the name of customer must be selected in the combobox. In all situations the total number of items is updated.

```
using System;
using
System.Collections.Generic;
using
System.ComponentModel;
using System.Data;
using
System.Drawing;
using System.Linq;
using System.Text;
using
System.Threading.Tasks;
using
System.Windows.Forms;

namespace Exercicio2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button2_Click(object sender, EventArgs e)
        {
            //Insere novo elemento na
            comboBox int cnt =
```

```
}  
  
private void button1_Click(object sender, EventArgs e)  
{  
    //Inicializa combobox  
    comboBox1.Items.Add("Ana")  
    ;  
    comboBox1.Items.Add("Paulo  
    ");  
    comboBox1.Items.Add("Pedro");  
    atualiza_itens();  
}  
  
private void button3_Click(object sender, EventArgs e)  
{  
    //Remove item da combobox  
    comboBox1.Items.Remove(comboBox1.SelectedItem);  
    atualiza_itens();  
}  
  
private void atualiza_itens()  
{  
    //Conta nº de itens da combobox  
    label2.Text = comboBox1.Items.Count.ToString();  
}  
}
```

5.6 Windows Forms: Working with Strings

A small scenario was considered to deal with strings. The image of this scenario is depicted in Figure 13. The layout of this scenario is composed of three elements:

- Two textboxes to insert two strings;
- A results multiline textbox that is used to write the results of the process;
- A “begin” button that is used to start the process

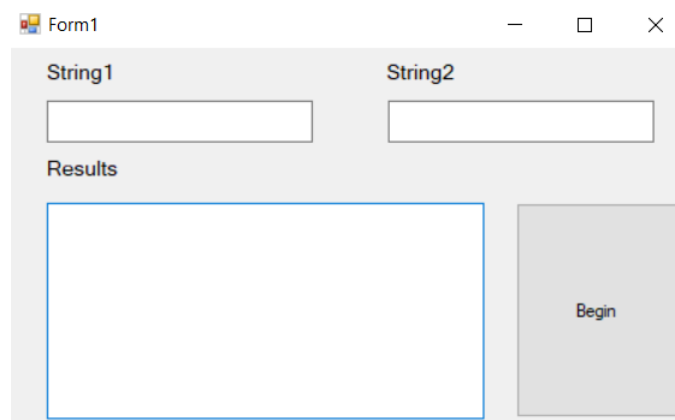


Figure 13 – Dealing with strings scenario

The program is composed of three phases:

1. Calculate the size of two given strings;
2. Verify if the second string is contained in the first string;
3. Verify if both strings are equal.

```
using System;
using
System.Collections.Generic;
using
System.ComponentModel;
using System.Data;
using
System.Drawing;
using System.Linq;
using System.Text;
using
System.Threading.Tasks;
using
System.Windows.Forms;

namespace Exercicio3
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

            //Calcular tamanho das
            stirngs          of  string1:  "  +  textBox2.Text.Length  +
            textBox1.AppendText("Size of string1: " + textBox2.Text.Length +
Environment.NewLine);
            stirngs          of  string2  "  +  textBox3.Text.Leng  +
            textBox1.AppendText("Size of string2: " + textBox3.Text.Length +
Environment.NewLine);

            //Comparar strings
            if (String.Compare(textBox2.Text, textBox3.Text) == 0)
            {
                textBox1.AppendText("The strings are equal." + Environment.NewLine);
            }
            else if (textBox2.Text.Contains(textBox3.Text))
            {
                textBox1.AppendText("String2 is contained in String1." +
Environment.NewLine);
            }
            else
            {
                textBox1.AppendText("The strings are different." +
Environment.NewLine);
            }
        }
    }
}
```

```
}  
}
```

5.7 Windows Forms: Interacting with databases

This scenario builds an application that interacts with a database. The layout of the scenario is depicted in Figure 14. Six functions are defined:

- Connect to a DB – establishes a new connection to a database. The database “Empresa” is composed only with two tables: (i) companies; and (ii) employees. An employee can only works in a company;
- Close DB – closes an established connection to the database;
- N° of companies – counts the number of companies that exist in the database;
- Name + salary – presents an XML document structure with all the employees’ names and salary;
- Insert company – inserts a new company in the database;
- Delete company – deletes the company previously inserted in the database.

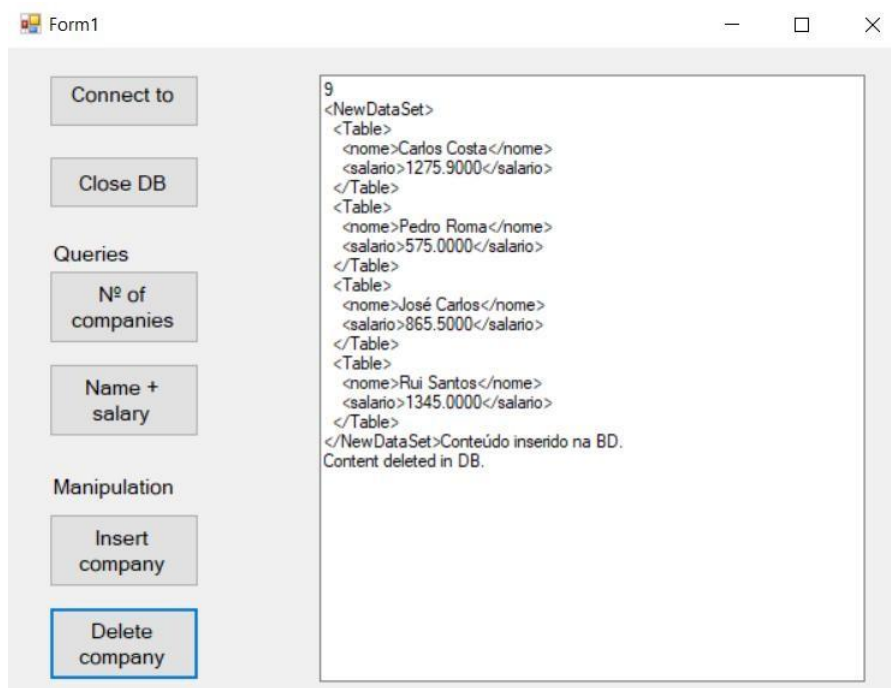


Figure 14 – Interacting with databases scenario

There are several alternatives to query data from a database. The first thing to consider is if the query will return just one or multiple values. If the query returns just one value, then the best approach is to use the “ExecuteScalar()” function. If it returns multiple values, there are two good solutions: (i) use a dataset and a SqlDataAdapter to return all the data directly to the dataset; or (ii) use a SqlDataReader, which implies that the connection to the database remains opened until there is data to be read. Both approaches are presented in the code, but the second approach is commented.

```
using System;
using
System.Collections.Generic;
using
System.ComponentModel;
using System.Data;
using
System.Drawing;
using System.Linq;
using System.Text;
using
System.Threading.Tasks;
using
System.Windows.Forms;
using
System.Data.SqlClient;
using System.IO;

namespace Exercicio5
{
    public partial class Form1 : Form
    {
        System.Data.SqlClient.SqlConnection con;

        public Form1()
        {
            InitializeComponent();
        }

        public int id_maximo;

        private void button1_Click(object sender, EventArgs e)
        {
            con = new System.Data.SqlClient.SqlConnection();

            //con.ConnectionString = @"Data Source=DESKTOP-DA31DSB\ferny;
                                Initial Catalog=Empresa; Integrated Security=True";
            //con.ConnectionString = @"Server = 192.168.106.187,1433; Database =
            Emulation; User ID = admin; Password = admin566751admin; Trusted_Connection =
            True";
            //con.ConnectionString = @"Server = NEKTON\SQLEXPRESS; Database =
            Emulation; User ID = admin; Password = admin566751admin; Trusted_Connection =
            True";
            con.ConnectionString = @"Server = DESKTOP-DA31DSB\SQLEXPRESS;
            Database = Empresa; Trusted_Connection = True";

            try
            {
                con.Open();
            }
            catch (SqlException)
            {
            }
```

```
//MessageBox.Show("Erro de ligação à BD.");  
MessageBox.Show("Erro de ligação à BD", "Erro",  
                MessageBoxButtons.OK, MessageBoxIcon.Error);  
return;
```



```
}

MessageBox.Show("Database Open");
}

private void button2_Click(object sender, EventArgs e)
{
    try
    {
        con.Close();
    }
    catch (SqlException)
    {
        return;
    }

    MessageBox.Show("Database Close");
}

private void button3_Click(object sender, EventArgs e)
{
    SqlCommand cmd = new SqlCommand();

    cmd.CommandText = "SELECT count(*) FROM
    Empresa"; cmd.CommandType =
    CommandType.Text; cmd.Connection = con;

    // Returns only one element
    Int32 count = (Int32)cmd.ExecuteScalar();

    textBox1.AppendText(count.ToString() + "\n");
}

private void button6_Click(object sender, EventArgs e)
{
    SqlCommand cmd = new SqlCommand();

    cmd.CommandText = "SELECT max(id) FROM
    Empresa"; cmd.CommandType =
    CommandType.Text; cmd.Connection = con;

    // Returns only one element
    string maximo = cmd.ExecuteScalar().ToString();

    //Update max value
    int v_id = Int32.Parse(maximo)
    + 1; id_maximo = v_id;
    string v_nome = "MyCompany";
```

```
string v_cidade = "Porto";

string sql_content = @"INSERT into Empresa (id, nome, cidade) VALUES
(@sql_id, @sql_nome, @sql_cidade)";

SqlCommand command = con.CreateCommand();

command.CommandType =
CommandType.Text;
command.CommandText = sql_content;
//Avoid SQL Injection
command.Parameters.Add(new SqlParameter("sql_id", v_id));
command.Parameters.Add(new SqlParameter("sql_nome",
v_nome)); command.Parameters.Add(new
SqlParameter("sql_cidade", v_cidade));

// Execute statement and returns a dataReader
SqlDataReader reader =
command.ExecuteReader();

string output = "Conteúdo inserido na
BD."; textBox1.AppendText(output +
"\n"); reader.Close();
}

private void button5_Click(object sender, EventArgs e)
{
    int v_id = id_maximo;

    string sql_content = "DELETE From Empresa Where id = @sql_id";

    SqlCommand command = con.CreateCommand();

    command.CommandType = CommandType.Text;
    command.CommandText = sql_content;
    command.Parameters.Add(new SqlParameter("sql_id",
v_id));

    // Executes statement and returns a
dataReader SqlDataReader reader =
command.ExecuteReader();

    string output = "Content deleted in
DB."; textBox1.AppendText(output +
"\n"); reader.Close();
}

private void button4_Click(object sender, EventArgs e)
{
    SqlCommand cmd = new SqlCommand();

    cmd.CommandText = "select nome, salario from
Colaborador"; cmd.CommandType =
```

```
CommandType.Text;  
cmd.Connection = con;
```

```
//Reading data from database to the
dataset DataSet dst = new DataSet();
SqlDataAdapter dap = new SqlDataAdapter(cmd.CommandText, con);
dap.Fill(dst);

//Writing data
var writer = new StringWriter();
dst.WriteXml(writer);
textBox1.AppendText(writer.ToString
());

/* Using datareader - low performance
// Reading contents
using (SqlDataReader reader = cmd.ExecuteReader())
{
    while (reader.Read())
    {
        for (int i = 0; i < reader.FieldCount; i++)
        {
            textBox1.AppendText(reader.GetValue(i).ToString());
        }
        textBox1.AppendText("\n");
    }
}
*/
}
}
```

5.8 Windows Forms: Interacting with XML files

This scenario intends to present a scenario in which we interact with XML files. The layout of the window is composed of the following elements:

- Four buttons – these buttons are used to: (i) list customers from a XML file; (ii) add a new customer to a XML file; (iii) delete a customer from a XML file; (iv) create a new file only with the customers that present a negative profit;
- Input data – it can be used by the user to write the code and name of a new customer. The customer to be deleted must also be equal to the code given in the textbox;
- Area field – write the appropriate messages to the customer, indicating which operation has been executed.

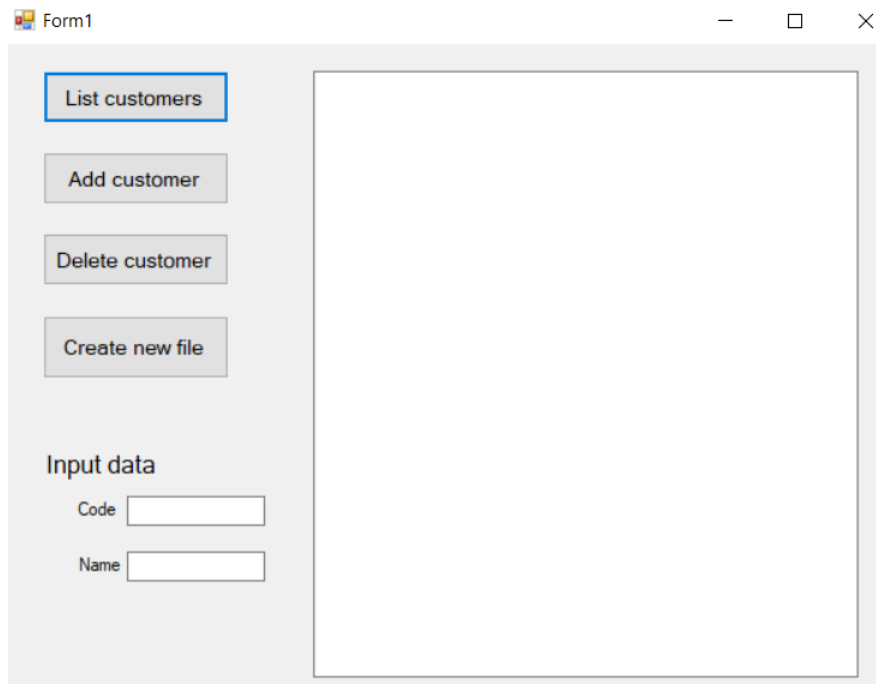


Figure 15 – Interacting with XML files scenario

It is also relevant to present the structure of “customer.xml” file. Each customer has information regarding his/her code, name and profit. The profit can be positive or negative. In the file below we have two customers with negative profit.

```
<?xml version="1.0" encoding="utf-8"?>
<customers>
  <customer cod="1" name="Mark" profit="1500" />
  <customer cod="2" name="Peter" profit="-500" />
  <customer cod="3" name="Tanya" profit="5500" />
  <customer cod="4" name="Rachel" profit="-3500" />
  <customer cod="5" name="Paul" profit="1200" />
</customers>
```

The source code of this scenario is given below. LINQ to XML was used to perform the requested operations. A “customer” class was created to receive the information from the XML file. In the last operation, a “customers2.xml” was created. This file is responsible to receive a copy of the customers with negative profit.

```
using System;
using
System.Collections.Generic;
using
System.ComponentModel;
using System.Data;
using
System.Drawing;
using System.Linq;
using System.Text;
using
```

```
using System.IO;
using System.Xml.Linq;

namespace Exercicio5
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            XElement xml =
            XElement.Load("Customers.xml"); foreach
            (XElement x in xml.Elements())
            {
                Customer c = new Customer();
                c.cod =
                int.Parse(x.Attribute("cod").Value);
                c.name = x.Attribute("name").Value;
                c.profit =
                float.Parse(x.Attribute("profit").Value);
                textBox1.AppendText(c.cod.ToString() + ";
                "); textBox1.AppendText(c.name + "; ");
                textBox1.AppendText(c.profit.ToString() +
                "\n");
            }
        }

        private void button2_Click(object sender, EventArgs e)
        {
            XElement x = new XElement("customer");
            x.Add(new XAttribute("cod",
            textBox2.Text)); x.Add(new
            XAttribute("name", textBox3.Text));
            x.Add(new XAttribute("profit", "0"));
            XElement xml =
            XElement.Load("Customers.xml");
            xml.Add(x);
            xml.Save("Customers.xml");
            textBox1.AppendText("New customer added.
            \n");
        }

        private void button3_Click(object sender, EventArgs e)
        {
            string cod = textBox2.Text;
            XElement xml = XElement.Load("Customers.xml");
            XElement x = xml.Elements().Where(p =>
            p.Attribute("cod").Value.Equals(cod)).First(); if (x != null)
            {
```

```
x.Remove();  
}  
xml.Save("Customers.xml");
```

```
        textBox1.AppendText("Customer deleted. \n");
    }

    private void button4_Click(object sender, EventArgs e)
    {
        int n = 0;
        XElement xml =
        XElement.Load("Customers.xml"); foreach
        (XElement x1 in xml.Elements())
        {
            Customer c = new Customer();
            c.cod =
            int.Parse(x1.Attribute("cod").Value);
            c.name = x1.Attribute("name").Value;
            c.profit = float.Parse(x1.Attribute("profit").Value);

            if (c.profit < 0)
            {
                XElement x2 = new XElement("customer");
                x2.Add(new XAttribute("cod", c.cod.ToString()));
                x2.Add(new XAttribute("name", c.name));
                x2.Add(new XAttribute("profit",
                c.profit.ToString()));

                XElement xml2 =
                XElement.Load("customers2.xml");
                xml2.Add(x2);
                xml2.Save("customers2.xml");
                n++;
            }
        }
        textBox1.AppendText("Number of customers in the file: "+n.ToString());
    }

    class Customer
    {
        public int cod;
        public string
        name; public float
        profit;
    }
}
```


Bibliography

Albahari, J. (2017). *C# 7.0 in a Nutshell: The Definitive Reference*. O'Reilly Media.

Arora, A. (2015, 01 13). *Using Await in Catch and Finally Blocks: A New Feature of C# 6.0*. Retrieved from <https://www.c-sharpcorner.com/UploadFile/16101a/using-await-in-catch-and-finally-block-a-new-feature-of-C-Sharp/>

Byahut, J. (2014, 06 23). *Create Xml File Using Linq to Xml*. Retrieved from ASP Helps: <http://www.asphelps.com/Linq/Create-xml.aspx>

David Grossman, G. F. (2013). *Computer Science Programming Basics*. O'Reilly Media.

Home & Learn. (2018, 07 12). *Visual C# .NET - Contents Page*. Retrieved from <https://www.homeandlearn.co.uk/csharp/csharp.html>

JavaTPoint. (2018, 07 12). *C#*. Retrieved from <https://www.javatpoint.com/csharp-data-types>

Kanjilal, J. (2015, 03 12). *Best practices in handling exceptions in C#*. Retrieved from <https://www.infoworld.com/article/2896294/application-development/best-practices-in-handling-exceptions-in-c.html>

Rodrigues, J. (2018, 07 12). *Manipulando arquivos XML em C#*. Retrieved from Linha deCodigo: <http://www.linhadecodigo.com.br/artigo/3449/manipulando-arquivos-xml-em-csharp.aspx>

Techopedia. (2017, 07 10). *Windows Forms*. Retrieved from <https://www.techopedia.com/definition/24300/windows-forms-net>

Troelsen, A., & Japikse, P. (2015). *C# 6.0 and the .NET 4.6 Framework*. Apress.

TutorialsPoint. (2018, 07 12). *C# Tutorial*. Retrieved from <https://www.tutorialspoint.com/csharp/index.htm>

Webber, Z. (2018). *C#: The Utmost Intermediate Course Guide In Fundamentals And Concept Of C# Programming*. Amazon Digital Services LLC.