LAB MANUAL

Computer Programming

COURSE CODE: CSC-221

Department of Software Engineering

Bahria University Karachi Campus

13 National Stadium Road

Preface

The course is designed to familiarize students with the basic structured programming skills. It emphasizes upon problem analysis, algorithm designing, and programs development and testing.

The objective of the course is to introduce a disciplined approach to Problem solving methods and algorithm development. The aim is to teach the syntax and vocabulary of C#. Key procedural programming topics like variables, arrays, strings, methods will be covered in detail. Simple programs will be coded, using logics, calculation and algorithm.

The language used for implementations is C#. In implementing the assignments, a good programming style is very important.

This guide is intended to be used by the students of Software Engineering Department of Bahria University Karachi Campus.

SE Department, Bahria University Karachi *Engr.Ramsha Mashood*

Table of Contents

Lab Manual for Computer Programming

Lab No. 1

PROGRAMMING BASICS

Objectives

To understand basic concept of computer programming.

LAB # 01

Programming Basics

Introduction

Any series of instructions to a computer to accomplish a task is a **Program**. The process of writing computer programs is known as **Programming** and **Programming languages** are used to write programs. It is set of commands that a computer has been "taught" to understand (Instructions must be written in a way the computer can understand).

All programming languages have certain features in common. For example:

- Variables
- Commands/Syntax (the way commands are structured)
- Loops
- Decisions
- Functions
-

Each programming language has a different set of rules about these features. To write a program we have to follow the following steps:

- Decide what steps are needed to complete the task
- Write the steps in pseudocode (written in English) or as a flowchart (graphic symbols)
- Translate into the programming language
- Try out the program and "debug" it (fix if necessary)

While doing programming we should know the basic points:

- void Main() must be present in all the programs
- { and } specify the beginning and the end of the program
- Initial statements are for the declaration of variables
- Statements that follow the declaration are the processing statements
- All statements should end with a semicolon

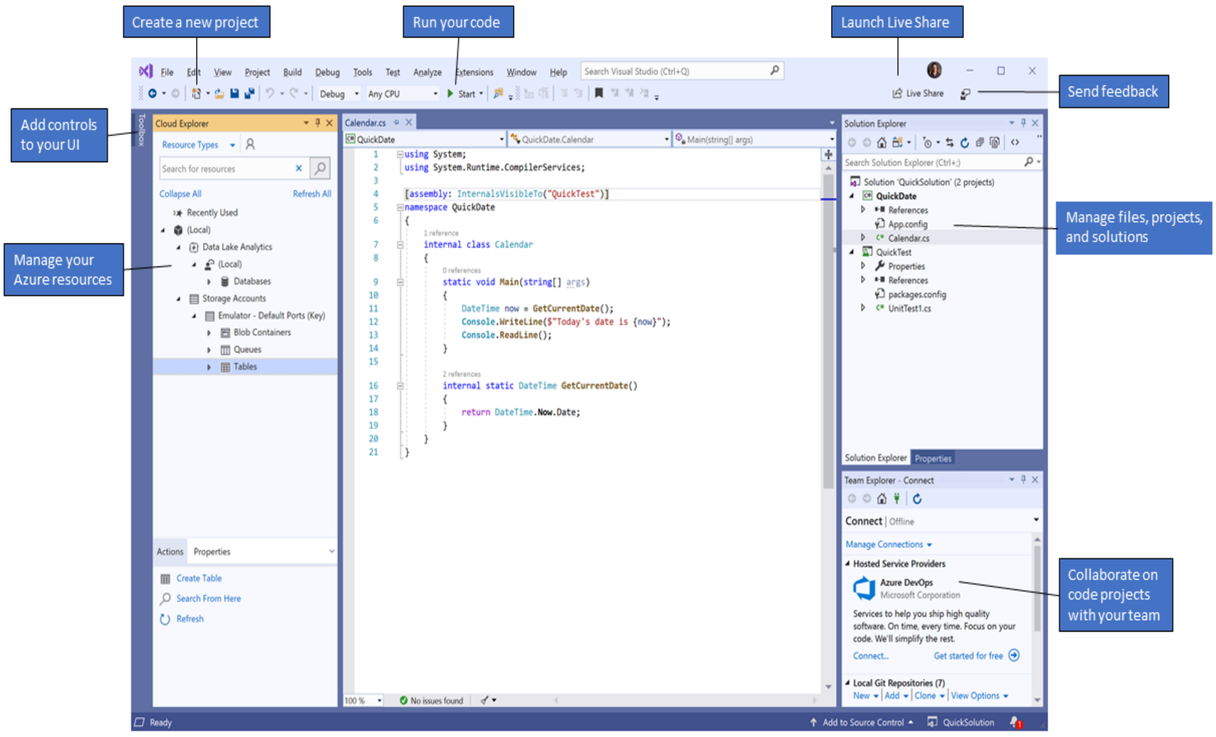
To Write the text representation of the specified objects, followed by the current line terminator, to the standard output stream using the specified format information. Following code is used: Console.WriteLine("");
Writes the text representation of the specified value or values to the standard output stream. Following code is used: Console.Write("");

Example Code:

- To write:
    1. Console.Write("First\nName");
    2. Console.Write("Last\tName");
- To write in single line:
    1. Console.WriteLine("First Name");

Introduction to VISUAL STUDIO IDE

The Visual Studio *integrated development environment* is a creative launching pad that you can use to edit, debug, and build code, and then publish an app. An integrated development environment (IDE) is a feature-rich program that can be used for many aspects of software development. Over and above the standard editor and debugger that most IDEs provide, Visual Studio includes compilers, code completion tools, graphical designers, and many more features to ease the software development process.


Create a new project
Run your code
Launch Live Share
Send feedback
Add controls to your UI
Manage your Azure resources
Manage files, projects, and solutions
Collaborate on code projects with your team

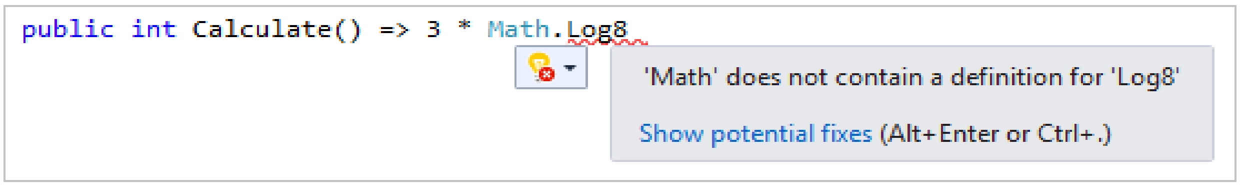
```
using System;
using System.Runtime.CompilerServices;

[assembly: InternalsVisibleTo("QuickTest")]
namespace QuickDate
{
    internal class Calendar
    {
        static void Main(string[] args)
        {
            DateTime now = GetCurrentDate();
            Console.WriteLine($"Today's date is {now}");
            Console.ReadLine();
        }

        internal static DateTime GetCurrentDate()
        {
            return DateTime.Now.Date;
        }
    }
}
```

CODE PRODUCTIVITY FEATURES

QUICK ACTIONS

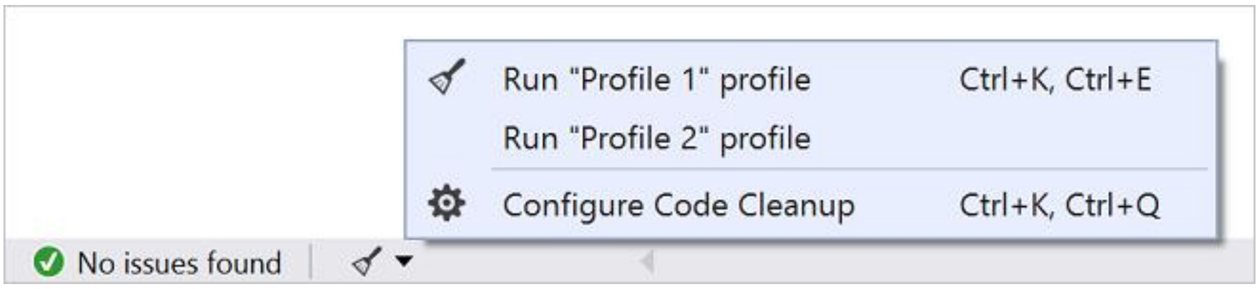


```
public int Calculate() => 3 * Math.Log8
```

'Math' does not contain a definition for 'Log8'

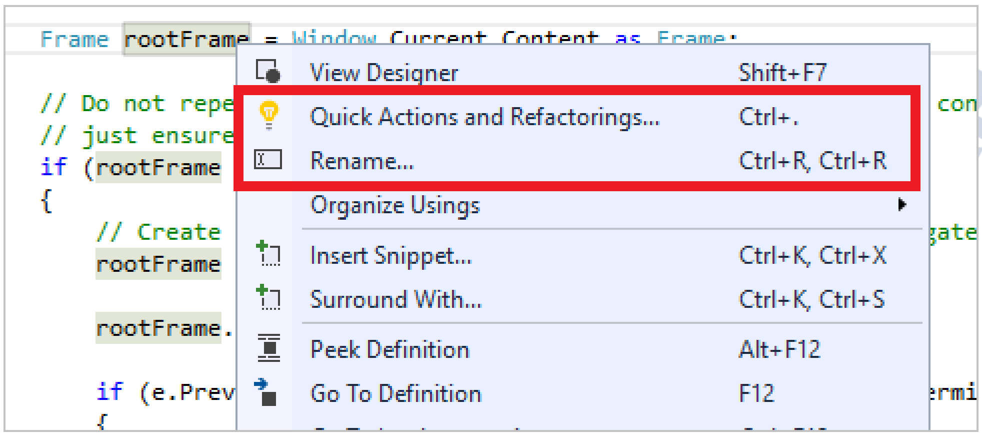
Show potential fixes (Alt+Enter or Ctrl+.)

CODE CLEANUP


Run "Profile 1" profile Ctrl+K, Ctrl+E
Run "Profile 2" profile
Configure Code Cleanup Ctrl+K, Ctrl+Q
No issues found

REFACTORING

Refactoring includes operations such as intelligent renaming of variables, extracting one or more lines of code into a new method, changing the order of method parameters, and more.



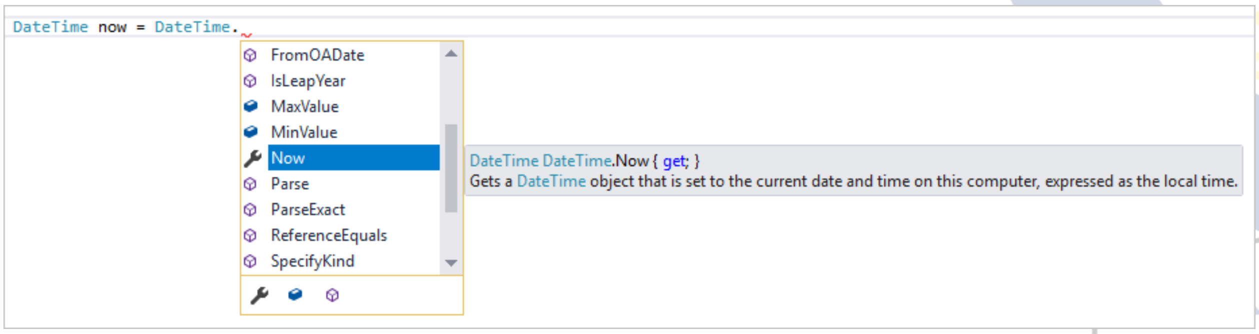
```
Frame rootFrame = Window.Current.Content as Frame;

// Do not repe
// just ensure
if (rootFrame
{
    // Create
    rootFrame

    rootFrame.

    if (e.Prev
    {
```

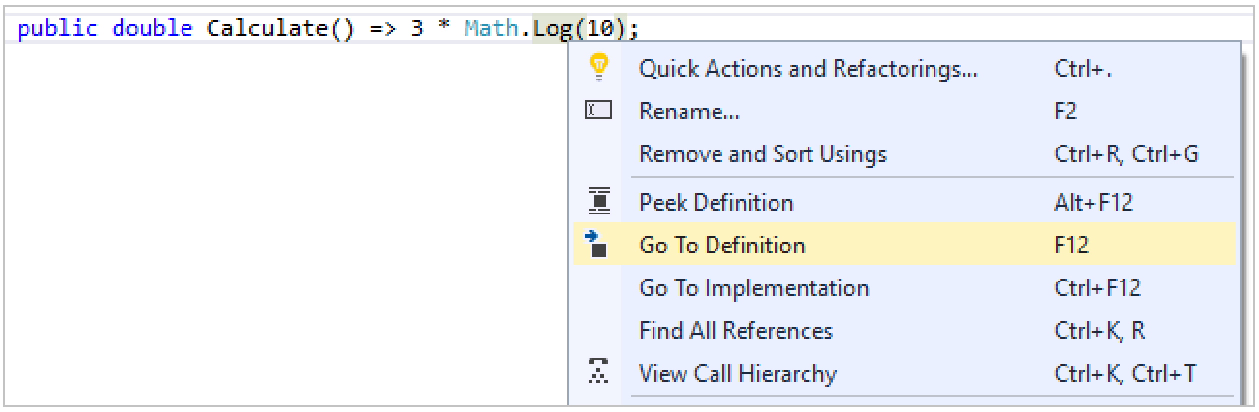
INTELLISENSE



```
DateTime now = DateTime.
```

DateTime DateTime.Now { get; }
Gets a DateTime object that is set to the current date and time on this computer, expressed as the local time.

GO TO DEFINITION



```
public double Calculate() => 3 * Math.Log(10);
```

Quick Actions and Refactorings... Ctrl+.
Rename... F2
Remove and Sort Usings Ctrl+R, Ctrl+G
Peek Definition Alt+F12
Go To Definition F12
Go To Implementation Ctrl+F12
Find All References Ctrl+K, R
View Call Hierarchy Ctrl+K, Ctrl+T

Escape sequence	Character represented
\a	Alert (Beep, Bell) (added in C89)
\b	Backspace
\e	escape character
\f	Form feed Page Break
\n	Newline (Line Feed)
\r	Carriage Return
\t	Horizontal Tab
\v	Vertical Tab
\\	Backslash
\'	Apostrophe or single quotation mark
\"	Double quotation mark
\?	Question mark (used to avoid trigraphs)

Time Boxing

Activity Name	Activity Time	Total Time
Login Systems + Setting up Visual Studio Environment	3 mints + 5 mints	8 mints
Walk through Theory & Tasks	60 mints	60 mints
Implement Tasks	80 mints	80 mints
Evaluation Time	30 mints	30 mints
	Total Duration	178 mints

Objectives/Outcomes

This Lab exercise delivers the idea/concept of:

- Basic programming concepts
- Implementation and working of output stream
- Basic primitive built-in Data-types in C#

Lab Tasks/Practical Work

1. Write step how to install visual studio
2. Show the output of all examples mention in the lab
3. Write a program to print your profile using c#.

Lab Manual for Computer Programming

Lab No. 2

Variables and arithmetic operations

Objectives

To understand variables and arithmetic operations .

LAB # 02

Variables and arithmetic operations

Primitive Built-in Types:

C# offer the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C# data types:

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void

The following table shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables.

Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647

short int	2bytes	-32768 to 32767
unsigned short int	Range	0 to 65,535
signed short int	Range	-32768 to 32767
long int	4bytes	-2,147,483,648 to 2,147,483,647
signed long int	4bytes	same as long int
unsigned long int	4bytes	0 to 4,294,967,295
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	2 or 4 bytes	1 wide character

The sizes of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

Operators:

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C# has rich set of built-in operators and some type of operators are listed below:

Arithmetic Operators

Relational Operators

Logical Operators

Assignment Operators

Arithmetic Operator:

Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
+	Adds two operands	A + B = 30
-	Subtracts second operand from the first	A - B = -10
*	Multiplies both operands	A * B = 200
/	Divides numerator by de-numerator	B / A = 2
%	Modulus Operator and remainder of after an integer division	B % A = 0

This lab will introduce one new topic, formatting of stream input. C# defines two standard streams that can be used for text mode input and output, respectively ReadLine() and WriteLine().

Using the standard input and output library, we will be able to interact with the user by printing messages on the screen and getting the user's input from the keyboard

C# uses a convenient abstraction called *streams* to perform input and output operations in sequential media such as the screen or the keyboard. A stream is an object where a program can either insert or extract characters to/from it. We do not really need to care about many specifications about the physical media associated with the stream - we only need to know it will accept or provide characters sequentially.

Read**Line** (returns a **string**): reads only single line from the standard input stream. As an example, it can be used to ask the user to enter their name or age.

Example Code 1:

```
static void Main(string[] args) {
string name;
//Variable for storing string value
Console.WriteLine("Please Enter Your Good Name");
//Accepting and holding values in name variable
name = Console.ReadLine();
//Displaying Output
Console.WriteLine("Welcome ” + name +” in your first csharp program");
//Holding console screen
 Console.ReadLine();
```

Example Code 2:

```
Console.WriteLine("Enter");
int ch1 = Console.Read();
int ch2 = Console.Read();
int ch3 = Console.Read();
Console.WriteLine(ch1);
Console.WriteLine(ch2);
Console.WriteLine(ch3);
Console.WriteLine(Convert.ToChar(ch1));
Console.WriteLine(Convert.ToChar(ch2));
Console.WriteLine(Convert.ToChar(ch3));
```

Type conversion

Type conversion is converting one type of data to another type. It is also known as Type Casting. In C#, type casting has two forms:

Implicit type conversion - These conversions are performed by C# in a type-safe manner. For example, are 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 the pre-defined functions. Explicit conversions require a cast operator.

C# Type Conversion Methods

S. No.	Methods & Description
1	**ToBoolean:** Converts a type to a Boolean value, where possible.
2	**ToChar:** Converts a type to a single Unicode character, where possible.
3	**ToDateTime:** Converts a type (integer or string type) to date-time structures.
4	**ToDecimal:** Converts a floating point or integer type to a decimal type.
5	**ToDouble:** Converts a type to a double type.
6	**ToInt16:** Converts a type to a 16-bit integer.
7	**ToInt32:** Converts a type to a 32-bit integer.
8	**ToInt64:**Converts a type to a 64-bit integer.
9	**ToSingle:** Converts a type to a small floating point number.
10	**ToString:** Converts a type to a string.
11	**ToType:** Converts a type to a specified type.
12	**ToUInt16:** Converts a type to an unsigned int type.
13	**ToUInt32:** Converts a type to an unsigned long type.
14	**ToUInt64:**Converts a type to an unsigned big integer.

Time Boxing

Activity Name	Activity Time	Total Time
Login Systems + Setting up Visual Studio Environment	3 mints + 5 mints	8 mints
Walk through Theory & Tasks	60 mints	60 mints
Implement Tasks	80 mints	80 mints
Evaluation Time	30 mints	30 mints
	Total Duration	178 mints

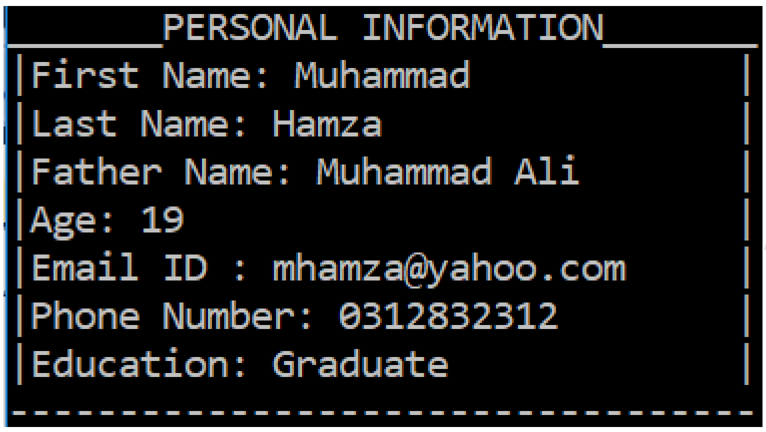
Objectives/Outcomes

This Lab exercise delivers the idea/concept of:

- Implementation and working of output stream
- Implementation and working of datatype conversion

Lab Tasks/Practical Work

1. Write a program to display your personal information. (Name, age, address, father's name, college name, NIC, phone number etc.) and display your marks sheet. ((Use Escape Sequences to create a formatted Output according to the given image).



```
_______PERSONAL INFORMATION_______
|First Name: Muhammad            |
|Last Name: Hamza                |
|Father Name: Muhammad Ali       |
|Age: 19                         |
|Email ID : mhamza@yahoo.com     |
|Phone Number: 0312832312        |
|Education: Graduate             |
----------------------------------
```

2. Write a C# program that displays the results of the expressions:
   – 3.0*5.0
   – 7.1*8.3-2.2
   – 3.2/ (6.1*5)
   – 15/4
   – 15%4
   – 5*3-(6*4)
3. Calculate the temperature in Celsius using **integer** values.
   – C = 5/9 * (F – 32)
4. Calculate the area of Circle.
5. Display the following results. (((a + b) * (c * e * d)) – e)/f
6. Display the result of the expression: (((a + b) * (c * e * d)) – e)/f
7. Write a program and print the output of first equation of the motion. For values take input from user. (vf=vi+at)

Lab Manual for Computer Programming

Lab No. 3

Formatted output

Objectives

To understand output stream functionality.

LAB # 03

Formatted output

WriteLine formatting in C#

WriteLine is used to output a line of string in C#. It is often required to output a integer, string or other variable in a certain way. We need to use formatting in such cases.
The format parameter in formatting is embedded with zero or more format specifications of the form

"{N [, M][: formatString]}", arg1, ... argN,

{index[,alignment][:formatString]}

where:

- N is a zero-based integer indicating the argument to be formatted.
- M is an optional integer indicating the width of the region to contain the formatted value, padded with spaces. If M is negative, the formatted value is left-justified; if M is positive, the value is right-justified.
- formatString is an optional string of formatting codes.
- argN is the expression to use at the equivalent position inside the quotes in the string.

Consider the following example to understand it

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {

            Console.WriteLine("{0,5} {1,5}", 123, 456);      // Right-aligned
            Console.WriteLine("{0,-5} {1,-5}", 123, 456);    // Left-aligned
            Console.WriteLine("{0,-10:D6} {1,-10:D6}", 123, 456); // D6 means 6 decimal digits
      Console.ReadLine();


      }
   }
}
```

Format Specifiers

Standard numeric format strings are used to return strings in commonly used formats. They take the form X0, in which X is the format specifier and 0 is the precision specifier. The format specifier can be one of the nine built-in format characters that define the most commonly used numeric format types, as shown in Table 10-1.

Table 10-1 *- String and WriteLine Format Specifiers*

Character	Interpretation
C or c	Currency
D or d	Decimal (decimal integer�don�t confuse with the .NET Decimal type)
E or e	Exponent
F or f	Fixed point
G or g	General
N or n	Currency
P or p	Percentage
R or r	Round-trip (for floating-point values only); guarantees that a numeric value converted to a string will be parsed back into the same numeric value
X or x	Hex

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {

        int i = 123456;
        Console.WriteLine("{0:C}", i); // $123,456.00
        Console.WriteLine("{0:D}", i); // 123456
        Console.WriteLine("{0:E}", i); // 1.234560E+005
        Console.WriteLine("{0:F}", i); // 123456.00
        Console.WriteLine("{0:G}", i); // 123456
        Console.WriteLine("{0:N}", i); // 123,456.00
        Console.WriteLine("{0:P}", i); // 12,345,600.00 %
        Console.WriteLine("{0:X}", i); // 1E240
        Console.ReadLine();


        }
    }
}
```

If you run above you should see the following command

```
$123,456.00
123456
1.234560E+005
123456.00
123456
123,456.00
12,345,600.00 %
1E240
```

To format output in C#, let us see examples to format date and double type.

Set formatted output for Double type.

```
using System;
class Demo {
  public static void Main(String[] args) {

    Console.WriteLine("Three decimal places...");
    Console.WriteLine(String.Format("{0:0.000}", 987.383));
    Console.WriteLine(String.Format("{0:0.000}", 987.38));
    Console.WriteLine(String.Format("{0:0.000}", 987.7899));

    Console.WriteLine("Thousands Separator...");
    Console.WriteLine(String.Format("{0:0,0.0}", 54567.46));
    Console.WriteLine(String.Format("{0:0,0}", 54567.46));
  }
}
```

Set formatted output for DateTime

```
using System;

static class Demo {

  static void Main() {


    DateTime d = new DateTime(2018, 2, 8, 12, 7, 7, 123);

    Console.WriteLine(String.Format("{0:y yy yyy yyyy}", d));

    Console.WriteLine(String.Format("{0:M MM MMM MMMM}", d));

    Console.WriteLine(String.Format("{0:d dd ddd dddd}", d));

  }

}
```

Printing concatenated string using Formatted String [Better Alternative]

A better alternative for printing concatenated string is using formatted string. Formatted string allows programmer to use placeholders for variables. For example,

The following line,

```
Console.WriteLine("Value = " + val);
```

can be replaced by,

```
Console.WriteLine("Value = {0}", val);
```

`{0}` is the placeholder for variable *val* which will be replaced by value of *val*. Since only one variable is used so there is only one placeholder.

Multiple variables can be used in the formatted string. We will see that in the example below.

Example 5: Printing Concatenated string using String formatting

```
using System;

namespace Sample
{
```

```
            class Test
            {
                        public static void Main(string[] args)
                        {
                                    int firstNumber = 5, secondNumber = 10, result;
                                    result = firstNumber + secondNumber;
                        Console.WriteLine("{0} + {1} = {2}", firstNumber, secondNumber, result);
                        }
            }
}
```

Time Boxing

Activity Name	Activity Time	Total Time
Login Systems + Setting up Visual Studio Environment	3 mints + 5 mints	8 mints
Walk through Theory & Tasks	60 mints	60 mints
Implement Tasks	80 mints	80 mints
Evaluation Time	30 mints	30 mints
	Total Duration	178 mints

Objectives/Outcomes

This Lab exercise delivers the idea/concept of:

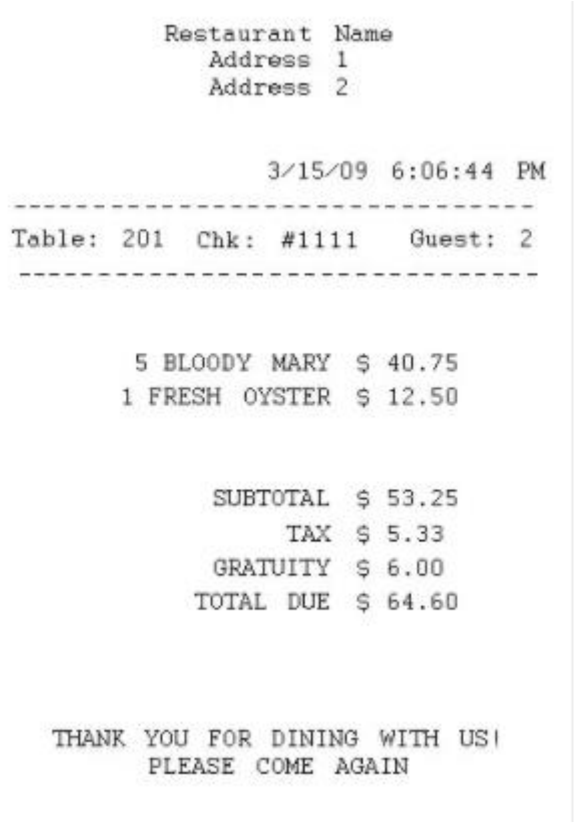
- Implementation and working of output stream
- Implementation and working of datatype conversion

Lab Tasks/Practical Work

1. Execute all example code and attach output in your lab file.
2. Create a small form which take following input from an employee of any company
    a. Name
    b. Father name
    c. Phone no
    d. Designation
    e. Date of hiring
    f. Salary

   Using above information display user profile in a formatted manner.

3. Create a simple c# program in which you will display the restaurant payment bill in formatted manner



```
           Restaurant  Name
              Address  1
              Address  2


                    3/15/09  6:06:44  PM
----------------------------------
Table:  201   Chk:  #1111    Guest:  2
----------------------------------


        5 BLOODY  MARY  $ 40.75
       1 FRESH  OYSTER  $ 12.50


            SUBTOTAL  $ 53.25
                 TAX  $ 5.33
            GRATUITY  $ 6.00
           TOTAL  DUE  $ 64.60


   THANK  YOU  FOR  DINING  WITH  US!
         PLEASE  COME  AGAIN
```

Lab Manual for Computer Programming

Lab No. 4

Operations and Expressions

Objectives

To understand basic concept, working and usage of operations and expressions

LAB # 04

Operation and Expression

Introduction

In this lab we will get acquainted with the operators in C# and the actions they can perform when used with the different data types. In the beginning, we will explain which operators have higher priority and we will analyze the different types of operators, according to the number of the arguments they can take and the actions they perform. In the second part, we will examine the conversion of data types. We will explain when and why it is needed to be done and how to work with different data types. At the end of the chapter, we will pay special attention to the expressions and how we should work with them. Finally, we have prepared exercises to strengthen our knowledge of the material in this chapter.

Operators

Every programming language uses operators, through which we can perform different actions on the data. Let's take a look at the operators in C# and see what they are for and how they are used.

Operators allow processing of primitive data types and objects. They take as an input one or more operands and return some value as a result. Operators in C# are special characters (such as "+", ".", "^", etc.) and they perform transformations on one, two or three operands. Examples of operators in C# are the signs for adding, subtracting, multiplication and division from math (+, -, *, /) and the operations they perform on the integers and the real numbers.

Operators in C#

Operators in C# can be separated in several different categories:

- **Arithmetic operators** – they are used to perform simple mathematical operations.
- **Assignment operators** – allow assigning values to variables.
- **Comparison operators** – allow comparison of two literals and/or variables.
- **Logical operators** – operators that work with Boolean data types and Boolean expressions.
- **Binary operators** – used to perform operations on the binary representation of numerical data.
- **Type conversion operators** – allow conversion of data from one type to another.

Operator Categories

Below is a list of the operators, separated into categories:

Category	Operators
arithmetic	-, +, *, /, %, ++, --
logical	&&, ||, !, ^
binary	&, |, ^, ~, <<, >>
comparison	==,!=, >, <, >=, <=
assignment	=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
string concatenation	+
type conversion	(type), as, is, typeof, sizeof
other	., new, (), [], ?:, ??

Operator Precedence in C#

Some operators have precedence (priority) over others. For example, in math multiplication has precedence over addition. The operators with a higher precedence are calculated before those with lower. The operator () is used to change the precedence and like in math, it is calculated first. The following table illustrates the precedence of the operators in C#:

Priority	Operators
Highest priority	(,)
	++, -- (as postfix), new, (type), typeof, sizeof
	++, -- (as prefix), +, - (unary), !, ~
	*, /, %
	+ (string concatenation)
	+, -
...	<<, >>
	<, >, <=, >=, is, as
	==, !=
	&, ^, |
Lowest priority	&&
	||
	?:, ??
	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=

The operators located upper in the table have higher precedence than those below them, and respectively they have an advantage in the calculation of an expression. To change the precedence of an operator we can use brackets. When we write expressions that are more complex or have many operators, it is recommended to use brackets to avoid difficulties in reading and understanding the code. For example:

```
// Ambiguous
x + y / 100

// Unambiguous, recommended
x + (y / 100)
```

Arithmetical Operators

The arithmetical operators in C# +, -, * are the same like the ones in math. They perform addition, subtraction and multiplication on numerical values and the result is also a numerical value.

The division operator / has different effect on integer and real numbers. When we divide an integer by an integer (like int, long and sbyte) the returned value is an integer (no rounding, the fractional part is cut). Such division is called an integer division. Example of integer division: 7 / 3 = 2.

Integer division by 0 is not allowed and causes a runtime exception DivideByZeroException. The remainder of integer division of integers can be obtained by the operator %. For example, 7 % 3 = 1, and –10 % 2 = 0. When dividing two real numbers or two numbers, one of which is real (e.g. float, double, etc.), a real division is done (not integer), and the result is a real number with a whole and a fractional part. For example: 5.0 / 2 = 2.5. In the division of real numbers it is allowed to divide by 0.0 and respectively the result is +∞ (Infinity), -∞ (-Infinity) or NaN (invalid value).

The operator for increasing by one (increment) ++ adds one unit to the value of the variable, respectively the operator -- (decrement) subtracts one unit from the value. When we use the operators ++ and -- as a prefix (when we place them immediately before the variable), the new value is calculated first and then the result is returned. When we use the same operators as post-fix (meaning when we place them immediately after the variable) the original value of the operand is returned first, then the addition or subtraction is performed. Here are some examples of arithmetic operators and their effect:

```
int squarePerimeter = 17;
double squareSide = squarePerimeter / 4.0;
double squareArea = squareSide * squareSide;
Console.WriteLine(squareSide); // 4.25
Console.WriteLine(squareArea); // 18.0625

int a = 5;
int b = 4;
Console.WriteLine(a + b);      // 9
Console.WriteLine(a + (b++));  // 9
Console.WriteLine(a + b);      // 10
Console.WriteLine(a + (++b));  // 11
Console.WriteLine(a + b);      // 11
Console.WriteLine(14 / a);     // 2
Console.WriteLine(14 % a);     // 4

int one = 1;
int zero = 0;
// Console.WriteLine(one / zero); // DivideByZeroException

double dMinusOne = -1.0;
double dZero = 0.0;
Console.WriteLine(dMinusOne / zero); // -Infinity
Console.WriteLine(one / dZero); // Infinity
```

Logical Operators

Logical (Boolean) operators take Boolean values and return a Boolean result (true or false). The basic Boolean operators are "AND" (&&), "OR" (||), "exclusive OR" (^) and logical negation (!). The following table contains the logical operators in C# and the operations that they perform:

x	y	!x	x && y	x || y	x ^ y
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

The following example illustrates the usage of the logical operators and their actions:

```
bool a = true;
bool b = false;
Console.WriteLine(a && b);              // False
Console.WriteLine(a || b);              // True
Console.WriteLine(!b);                  // True
Console.WriteLine(b || true);           // True
Console.WriteLine((5 > 7) ^ (a == b));  // False
```

Comparison Operators

Comparison operators in C# are used to compare two or more operands. C# supports the following comparison operators:

- greater than (>)
- less than (<)
- greater than or equal to (>=)
- less than or equal to (<=)
- equality (==)
- difference (!=)

All comparison operators in C# are binary (take two operands) and the returned result is a Boolean value (true or false). Comparison operators have lower priority than arithmetical operators but higher than the assignment operators. The following example demonstrates the usage of comparison operators in C#:

```
int x = 10, y = 5;
Console.WriteLine("x > y : " + (x > y));     // True
Console.WriteLine("x < y : " + (x < y));     // False
Console.WriteLine("x >= y : " + (x >= y));   // True
Console.WriteLine("x <= y : " + (x <= y));   // False
Console.WriteLine("x == y : " + (x == y));   // False
Console.WriteLine("x != y : " + (x != y));   // True
```

Assignment Operators

The operator for assigning value to a variable is "=" (the character for mathematical equation). The syntax used for assigning value is as it follows:

operand1 = literal, expression or operand2;

Here is an example to show the usage of the assignment operator:

```
int x = 6;
string helloString = "Hello string.";
int y = x;
```

Cascade Assignment

The assignment operator can be used in cascade (more than once in the same expression). In this case assignments are carried out consecutively from right to left. Here’s an example:

```
int x, y, z;

x = y = z = 25;
```

Compound Assignment

Operators Except the assignment operator there are also compound assignment operators. They help to reduce the volume of the code by typing two operations together with an operator: operation and assignment. Compound operators have the following syntax: Here is an example of a compound operator for assignment:

```
int x = 2;
int y = 4;

x *= y; // Same as

Console.WriteLine(x); // 8

x = x * y;

Console.WriteLine(x); // 8
```

Time Boxing

Activity Name	Activity Time	Total Time
Login Systems + Setting up Visual Studio Environment	3 mints + 5 mints	8 mints
Walk through Theory & Tasks	60 mints	60 mints
Implement Tasks	80 mints	80 mints
Evaluation Time	30 mints	30 mints
	Total Duration	178 mints

Objectives/Outcomes

This Lab exercise delivers the idea/concept of:

- Implementation of operations
- Creation and implementation of expressions.

Lab Tasks/Practical Work

1. Which of the following values can be assigned to variables of type float, double and decimal: 5, -5.01, 34.567839023; 12.345; 8923.1234857; 3456.091124875956542151256683467?
2. Create a simple calculator which will perform all arithmetical, Bit wise operation and logical operation on two number
3. Create a simple program to calculate Hypotenuse using Pythagoras theorem c^2 =(a^2 + b^2)

Lab Manual for Computer Programming

Lab No. 5

Decision Construct

Objectives

To understand basic concept, working and usage of Decision Construct

LAB # 05

Introduction to Decision Construct

Introduction

In this lab we will explain the syntax of the conditional operators if and if-else with suitable examples. We will focus on the best practices to be followed to achieve a better programming style when using nested or other types of conditional statements.

Conditional Statements "if" and "if-else"

After reviewing how to compare expressions, we will continue with conditional statements, which will allow us to implement programming logic. Conditional statements if and if-else are conditional control statements. Because of them the program can behave differently based on a defined condition checked during the execution of the statement.

Conditional Statement "if"

The main format of the conditional statements if is as follows:

It includes: if-clause, Boolean expression and body of the conditional statement.

```
if (Boolean expression)
{
   Body of the conditional statement;
}
```

The Boolean expression can be a Boolean variable or Boolean logical expression. Boolean expressions cannot be integer (unlike other programming languages like C and C++). The body of the statement is the part locked between the curly brackets: {}. It may consist of one or more operations (statements). When there are several operations, we have a complex block operator, i.e. series of commands that follow one after the other, enclosed in curly brackets. The expression in the brackets which follows the keyword if must return the Boolean value true or false. If the expression is calculated to the value true, then the body of a conditional statement is executed. If the result is false, then the operators in the body will be skipped.

Conditional Statement "if" – Example

Let's take a look at an example of using a conditional statement if:

```
static void Main()
{
  Console.WriteLine("Enter two numbers.");
  Console.Write("Enter first number: ");
  int firstNumber = int.Parse(Console.ReadLine());
  Console.Write("Enter second number: ");
  int secondNumber = int.Parse(Console.ReadLine());
  int biggerNumber = firstNumber;
  if (secondNumber > firstNumber)
  {
    biggerNumber = secondNumber;
  }
  Console.WriteLine("The bigger number is: {0}", biggerNumber);
}
```

If we start the example and enter the numbers 4 and 5 we will get the following result:

```
Enter two numbers.
Enter first number: 4
Enter second number: 5
The bigger number is: 5
```

Conditional Statement "if-else"

In C#, as in most of the programming languages there is a conditional statement with else clause: the if-else statement. Its format is the following:

```
if (Boolean expression)
{
  Body of the conditional statement;
}
else
{
  Body of the else statement;
}
```

The format of the if-else structure consists of the reserved word if, Boolean expression, body of a conditional statement, reserved word else and else-body statement. The body of else-structure may consist of one or more operators, enclosed in curly brackets, same as the body of a conditional statement. This statement works as follows: the expression in the brackets (a Boolean expression) is calculated. The calculation result must be Boolean – true or false. Depending on the result there are two possible outcomes. If the Boolean expression is calculated to true, the body of the conditional statement is executed and the else-statement is omitted and its operators do not execute. Otherwise, if the Boolean expression is calculated to false, the else-body is executed, the main body of the conditional statement is omitted and the operators in it are not executed.

Conditional Statement "if-else" - Example

Let’s take a look at the next example and illustrate how the if-else statement works:

```
static void Main()
{
   int x = 2;
   if (x > 3)
   {
      Console.WriteLine("x is greater than 3");
   }
   else
   {
      Console.WriteLine("x is not greater than 3");
   }
}
```

The program code can be interpreted as follows: if x>3, the result at the end is: "x is greater than 3", otherwise (else) the result is: "x is not greater than 3". In this case, since x=2, after the calculation of the Boolean expression the operator of the else structure will be executed. The result of the example is:

```
x is not greater than 3
```

Nested "if" Statements

Sometimes the programming logic in a program or an application needs to be represented by multiple if-structures contained in each other. We call them nested if or nested if-else structures. We call nesting the placement of an if or if-else structure in the body of another if or else structure. In such situations every else clause corresponds to the closest previous if clause. This is how we understand which else clause relates to which if clause. It’s not a good practice to exceed three nested levels, i.e. we should not nest more than three conditional statements into one another. If for some reason we need to nest more than three structures, we should export a part of the code in a separate method (see chapter Methods).

Nested "if" Statements - Example

Here is an example of using nested if structures:

```
int first = 5;
int second = 3;

if (first == second)
{
  Console.WriteLine("These two numbers are equal.");
}
else
{
  if (first > second)
  {
    Console.WriteLine("The first number is greater.");
  }
  else
  {
    Console.WriteLine("The second number is greater.");
  }
}
```

In the example above we have two numbers and compare them in two steps: first we compare whether they are equal and if not, we compare again, to determine which one is the greater. Here is the result of the execution of the code above:

```
The first number is greater.
```

Sequences of "if-else-if-else-…"

Sometimes we need to use a sequence of if structures, where the else clause is a new if structure. If we use nested if structures, the code would be pushed too far to the right. That’s why in such situations it is allowed to use a new if right after the else. It’s even considered a good practice. Here is an example:

```
char ch = 'X';
if (ch == 'A' || ch == 'a')
{
  Console.WriteLine("Vowel [ei]");
}
else if (ch == 'E' || ch == 'e')
{
  Console.WriteLine("Vowel [i:]");
}
else if (ch == 'I' || ch == 'i')
{
  Console.WriteLine("Vowel [ai]");
}
else if (ch == 'O' || ch == 'o')
{
  Console.WriteLine("Vowel [ou]");
}
else if (ch == 'U' || ch == 'u')
{
  Console.WriteLine("Vowel [ju:]");
}
else
{
  Console.WriteLine("Consonant");
}
```

The program in the example makes a series of comparisons of a variable to check if it is one of the vowels from the English alphabet. Every following comparison is done only in case that the previous comparison was not true. In the end, if none of the if-conditions is not fulfilled, the last else clause is executed. Thus, the result of the example is as follows:

```
Consonant
```

Conditional "if" Statements – Good Practices

Here are some guidelines, which we recommend for writing if, structures:

- Use blocks, surrounded by curly brackets {} after if and else in order to avoid ambiguity
- Always format the code correctly by offsetting it with one tab inwards after if and else, for readability and avoiding ambiguity.
- Prefer switch-case structure to of a series of if-else-if-else-…
- structures or nested if-else statement, if possible.

Conditional Statement "switch-case"

In the following section we will cover the conditional statement switch. It is used for choosing among a list of possibilities.

The structure switch-case chooses which part of the programming code to execute based on the calculated value of a certain expression (most often of integer type). The format of the structure for choosing an option is as follows:

```
switch (integer_selector)
{
  case integer_value_1:
    statements;
    break;
  case integer_value_2:
    statements;
    break;
  // …
  default:
    statements;
    break;
}
```

The selector is an expression returning a resulting value that can be compared, like a number or string. The switch operator compares the result of the selector to every value listed in the case labels in the body of the switch structure. If a match is found in a case label, the corresponding structure is executed (simple or complex). If no match is found, the default statement is executed (when such exists). The value of the selector must be calculated before comparing it to the values inside the switch structure. The labels should not have repeating values, they must be unique. As it can be seen from the definition above, every case ends with the operator break, which ends the body of the switch structure. The C# compiler requires the word break at the end of each case-section containing code. If no code is found after a case-statement, the break can be omitted and the execution passes to the next case-statement and continues until it finds a break operator. After the default structure break is obligatory. It is not necessary for the default clause to be last, but it's recommended to put it at the end, and not in the middle of the switch structure.

Rules for Expressions in Switch

The switch statement is a clear way to implement selection among many options (namely, a choice among a few alternative ways for executing the code). It requires a selector, which is calculated to a certain value. The selector type could be an integer number, char, string or enum. If we want to use for example an array or a float as a selector, it will not work. For noninteger data types, we should use a series of if statements.

Using Multiple Labels

Using multiple labels is appropriate, when we want to execute the same structure in more than one case. Let’s look at the following example:

```
int number = 6;
switch (number)
{
   case 1:
   case 4:
   case 6:
   case 8:
   case 10:
      Console.WriteLine("The number is not prime!"); break;
   case 2:
   case 3:
   case 5:
   case 7:
      Console.WriteLine("The number is prime!"); break;
   default:
      Console.WriteLine("Unknown number!"); break;
}
```

In the above example, we implement multiple labels by using case statements without break after them. In this case, first the integer value of the selector is calculated – that is 6, and then this value is compared to every integer value in the case statements. When a match is found, the code block after it is executed. If no match is found, the default block is executed.

Good Practices When Using "switch-case"

- A good practice when using the switch statement is to put the default statement at the end, in order to have easier to read code.
- It’s good to place first the cases, which handle the most common situations. Case statements, which handle situations occurring rarely, can be placed at the end of the structure.
- If the values in the case labels are integer, it’s recommended that they be arranged in ascending order.
- If the values in the case labels are of character type, it’s recommended that the case labels are sorted alphabetically.
- It’s advisable to always use a default block to handle situations that cannot be processed in the normal operation of the program. If in the normal operation of the program the default block should not be reachable, you could put in it a code reporting an error.

Time Boxing

Activity Name	Activity Time	Total Time
Login Systems + Setting up Visual Studio Environment	3 mints + 5 mints	8 mints
Walk through Theory & Tasks	60 mints	60 mints
Implement Tasks	80 mints	80 mints
Evaluation Time	30 mints	30 mints
	Total Duration	178 mints

Objectives/Outcomes

This Lab exercise delivers the idea/concept of:

- Creation and implementation of decision construct.

Lab Tasks/Practical Work

1. Create a simple program to check whether the no is even or odd.
2. Create a simple calculator using if else condition in which you will take two number as input and ask user which operation he/she wants to perform.
3. Write a Program using nested ifs in which the user is asked to do the following:
    a. Confirm if user wants to play the game “Guess the secret number”
    b. If false, then print message “Maybe next time”; if true, then ask “Enter your age”
    c. If age is less than 5 then print “You are too young to play”, else ask “Enter any number”.
    d. If number is equal to the secret number then print “You have successfully guessed the secret number; else print “You were not successful in guessing the secret number”.

    Note: You have to store a secret number in a variable in order to compare it.
4. Create simple application which will check the vowel using switch case
5. Create a simple program for bahria check in system

Lab Manual for Computer Programming

Lab No. 6

Introduction to Loops

Objectives

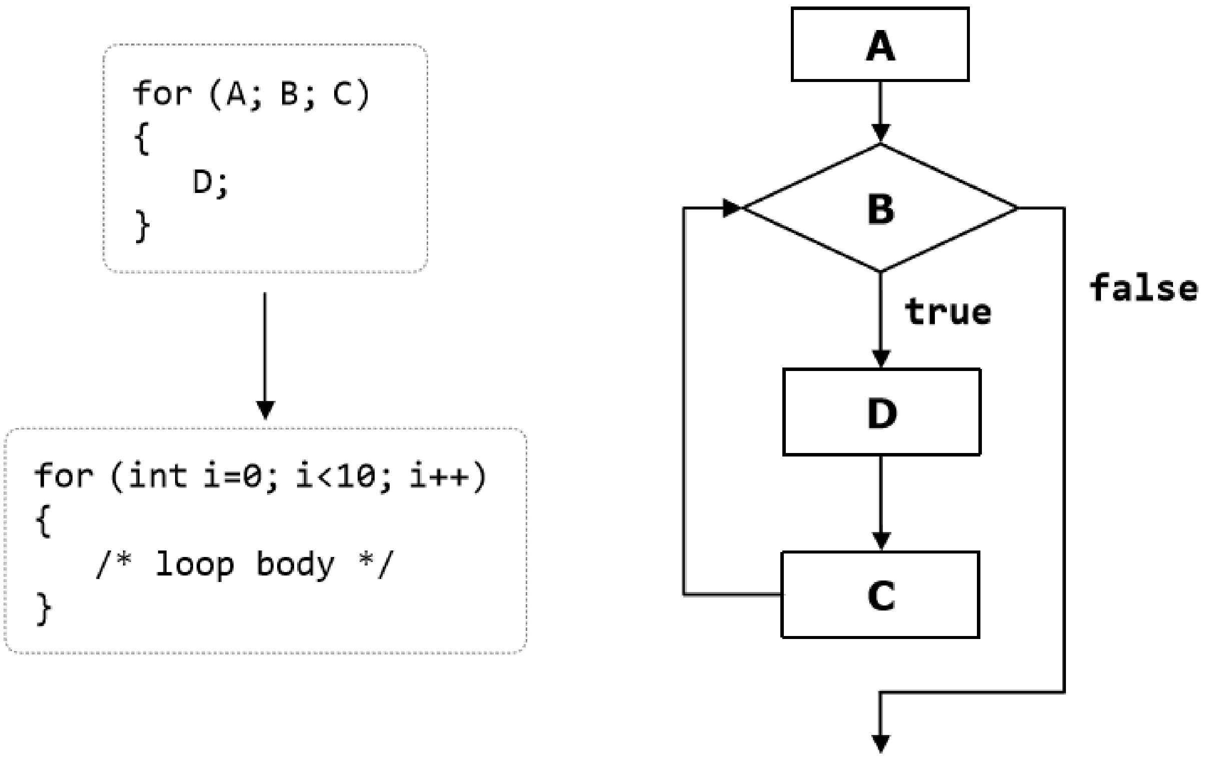
To understand basic concept, implementation and usage of loops.

LAB # 06

Introduction to LOOPS

For Loops

For-loops are a slightly more complicated than while and do-while loops but on the other hand they can solve more complicated tasks with less code. Here is the scheme describing for-loops:



```
for (A; B; C)
{
    D;
}
```

```
for (int i=0; i<10; i++)
{
    /* loop body */
}
```

They contain an initialization block (A), condition (B), body (D) and updating commands for the loop variables (C). We will explain them in details shortly. Before that, let's look at how the program code of a for-loop looks like:

```
for (initialization; condition; update)
{
  loop's body;
}
```

It consists of an initialization part for the counter (in the pattern int i = 0), a Boolean condition (i < 10), an expression for updating the counter (i++, it might be i-- or for instance, i = i + 3) and body of the loop.

The counter of the loop distinguishes it from other types of loops. Most often the counter changes from a given initial value to a final one in ascending order, for example from 1 to 100. The number of iterations of a given forloop is usually known before its execution starts. A for-loop can have one or several loop variables that move in ascending or descending order or with a step. It is possible one loop variable to increase and the other – to decrease. It is even possible to make a loop from 2 to 1024 in steps of multiplication by 2, since the update of the loop variables can contain not only addition, but any other arithmetic (as well as other) operations.

Since none of the listed elements of the for-loops is mandatory, we can skip them all and we will get an infinite loop:

```
for ( ; ; )
{
   // Loop body
}
```

Now let’s consider in details the separate parts of a for-loop.

Initialization of For Loops

For-loops can have an initialization block:

```
for (int num = 0; …; …)
{
   // The variable num is visible here and it can be used
}
// Here num can not be used
```

For-loops can have an **initialization block**:

```
for (int num = 0; …; …)
{
   // The variable num is visible here and it can be used
}
// Here num can not be used
```

It is executed only once, just before entering the loop. Usually the initialization block is used to declare the counter-variable (also called a loop variable) and to set its initial value. This variable is "visible" and can be used only within the loop. In the initialization block is possible to declare and initialize more than one variable.

For-loops can have a **loop condition**:

```
for (int num = 0; num < 10; …)
{
   // Loop body
}
```

The condition (loop condition) is evaluated once before each iteration of the loop, just like in the while loops. For result true the loop's body is executed, for result false it is skipped and the loop ends (the program continues immediately after the last line of the loop's body).

The last element of a for-loop contains **code that updates** the loop variable:

```
for (int num = 0; num < 10; num++)
{
   // Loop body
```

}

This code is executed at each iteration, after the loop's body has been executed. It is most commonly used to update the value of the countervariable.

The body of the loop contains a block with source code. The loop variables, declared in the initialization block of the loop are available in it.

Here is a complete **example** of a for-loop:

```
for (int i = 0; i <= 10; i++)
{
   Console.Write(i + " ");
}
```

The result of its execution is the following:

```
0 1 2 3 4 5 6 7 8 9 10
```

Here is another, more complicated example of a for-loop, in which we have two variables i and sum, that initially have the value of 1, but we update them consecutively at each iteration of the loop:

```
for (int i = 1, sum = 1; i <= 128; i = i * 2, sum += i)
{
  Console.WriteLine("i={0}, sum={1}", i, sum);
}
```

The result of this loop's execution is the following:

```
i=1, sum=1
i=2, sum=3
i=4, sum=7
i=8, sum=15
i=16, sum=31
i=32, sum=63
i=64, sum=127
i=128, sum=255
```

For-Loop with Several Variables

As we have already seen, in the construct of a for-loop we can use multiple variables at the same time. Here is an example in which we have two counters. One of the counters moves up from 1 and the other moves down from 10:

```
for (int small=1, large=10; small<large; small++, large--)
{
  Console.WriteLine(small + " " + large);
}
```

The condition for loop termination is overlapping of the counters. Finally we get the following result:

```
1 10
2 9
```

```
3 8
4 7
5 6
```

Operator "continue"

The continue operator stops the current iteration of the inner loop, without terminating the loop. With the following example we will examine how to use this operator. We will calculate the sum of all odd integers in the range [1…n], which are not divisible by 7 by using the for-loop:

```
int n = int.Parse(Console.ReadLine());
int sum = 0;
for (int i = 1; i <= n; i += 2)
{
   if (i % 7 == 0)
   {
      continue;
   }
   sum += i;
}
Console.WriteLine("sum = " + sum);
```

First we initialize the loop's variable with a value of 1 as this is the first odd integer within the range [1…n]. After each iteration of the loop we check if i has not yet exceeded n (i <= n). In the expression for updating the variable we increase it by 2 in order to pass only through the odd numbers. Inside the loop body we check whether the current number is divisible by 7. If so we call the operator continue, which skips the rest of the loop's body (it skips adding the current number to the sum). If the number is not divisible by seven, it continues with updating of the sum with the current number.

In this lab we will also examine the nested loops, these are programming constructs consisting of several loops located into each other. The innermost loop is executed more times, and the outermost – less times. Let's see how two nested loops look like:

```
for (initialization, verification, update) {
        for (initialization, verification, update)  {
                executable code
        }
… }
```

After initialization of the first for loop, the execution of its body will start, which contains the second (nested) loop. Its variable will be initialized, its condition will be checked and the code within its body will be executed, then the variable will be updated and execution will continue until the condition returns false. After that the second iteration of the first for loop will continue, its variable will be updated and the whole second loop will be performed once again. The inner loop will be fully executed as many times as the body of the outer loop.

Let's solve the following problem: for a given number n, to print on the console a triangle with n number of lines, looking like this:

```
1
1 2
1 2 3
…
1 2 3 … n
```

We will solve the problem with two for-loops. The outer loop will traverse the lines, and the inner one – the elements in them. When we are on the first line, we have to print "1" (1 element, 1 iteration of the inner loop). On the second line we have to print "1 2" (2 elements, 2 iterations of the internal loop). We see that there is a correlation between the line on which we are and the number of the elements that we print. This tells us how to organize the inner loop’s structure:

- We initialize the loop variable with 1 (the first number that we will print): col = 1;
- The repetition condition depends on the line on which we are: col <= row;
- We increase the loop variable with one unit at each iteration of the internal loop.

Basically, we need to implement a for-loop (external) from 1 to n (for the lines) and put another for-loop (internal) in it – for the numbers on the current line, which should spin from 1 to the number of the current line. The external loop should go through the lines while the internal – through the columns of the current line. Finally, we get the following code:

```
int n = int.Parse(Console.ReadLine());
for (int row = 1; row <= n; row++)
{
   for (int col = 1; col <= row; col++)
   {
      Console.Write(col + " ");
   }
   Console.WriteLine();
}
```

If we execute it, we will make sure that it works correctly. Here is how the result for n=7 looks like:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
```

Prime Numbers in an Interval – Example

Let’s consider another example of nested loops. We set a goal to print on the console all prime number in the interval [n…m]. We will limit the interval by a for-loop and in order to check for a prime number we will use a nested while loop:

```
Console.Write("n = ");
int n = int.Parse(Console.ReadLine());
Console.Write("m = ");
int m = int.Parse(Console.ReadLine());

for (int num = n; num <= m; num++) {
        bool prime = true;
        int divider = 2;
        int maxDivider = (int)Math.Sqrt(num);
        while (divider <= maxDivider)  {
                if (num % divider == 0)   {
                prime = false;
                break;
                }
                divider++;
        }
        if (prime)  {
        Console.Write(" " + num);
    }    }
```

Time Boxing

Activity Name	Activity Time	Total Time
Login Systems + Setting up Visual Studio Environment	3 mints + 5 mints	8 mints
Walk through Theory & Tasks	60 mints	60 mints
Implement Tasks	80 mints	80 mints
Evaluation Time	30 mints	30 mints
	Total Duration	178 mints

Objectives/Outcomes

This exercise delivers the idea/ concept of:

- Creation and implementation of for.

Lab Tasks/Practical Work

1. Cube series without using power maths function. (Use For loop)
2. Square Series without using power maths function (use For loop)
3. Repeatedly print the value of the variable xValue, decreasing it by 0.5 each time, as long as xValue remains positive.
4. Print the square roots of the first 25 odd positive integers.
5. Make a game in C#, in which give 5 tries to the user to guess the value of the number.
6. Generate Stars using 2 for loops

```
*
**
***
****
*****
******
```

7. Write a program that reads from the console a positive integer number N (N < 20) and prints a matrix of numbers as on the figures below:

N = 3

1	2	3
2	3	4
3	4	5

N = 4

1	2	3	4
2	3	4	5
3	4	5	6
4	5	6	7

Lab Manual for Computer Programming

Lab No. 7

While LOOPS

Objectives

To understand basic concept, implementation and usage of while loops.

LAB # 07

While loops

1. Loops

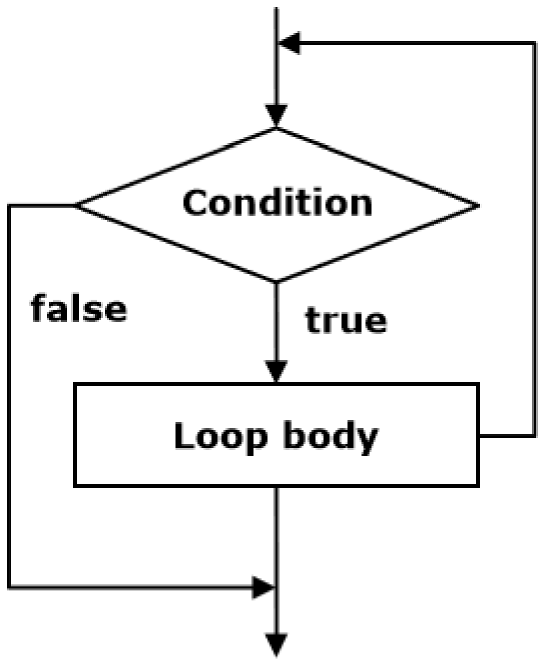
In programming often requires repeated execution of a sequence of operations. A loop is a basic programming construct that allows repeated execution of a fragment of source code. Depending on the type of the loop, the code in it is repeated a fixed number of times or repeats until a given condition is true (exists). Loops that never end are called infinite loops. Using an infinite loop is rarely needed except in cases where somewhere in the body of the loop a break operator is used to terminate its execution prematurely.

a. While Loop:

One of the simplest and most commonly used loops is while.

```
while (condition)
{
   loop body;
}
```

In the code above example, condition is any expression that returns a Boolean result – true or false. It determines how long the loop body will be repeated and is called the loop condition. In this example the loop body is the programming code executed at each iteration of the loop, i.e. whenever the input condition is true. The behavior of while loops can be represented by the following scheme:



In the while loop, first of all the Boolean expression is calculated and if it is true the sequence of operations in the body of the loop is executed. Then again, the input condition is checked and if it is true again the body of the loop is executed. All this is repeated again and again until at some point the conditional expression returns value false. At this point the loop stops and the program continues to the next line, immediately after the body of the loop. The body of the while loop may not be executed even once if in the beginning the condition of the cycle returns false. If the condition of the cycle is never broken the loop will be executed indefinitely.

Let's consider a very simple example of using the while loop. The purpose of the loop is to print on the console the numbers in the range from 0 to 9 in ascending order:

```
// Initialize the counter
int counter = 0;
// Execute the loop body while the loop condition holds
while (counter <= 9)
{
   // Print the counter value
   Console.WriteLine("Number : " + counter);
   // Increment the counter
   counter++;
}
```

Example 2:

```
Console.Write("n = ");
int n = int.Parse(Console.ReadLine());
int num = 1;
int sum = 1;
Console.Write("The sum 1");
while (num < n)
{
   num++;
   sum += num;
   Console.Write(" + " + num);
}
Console.WriteLine(" = " + sum);
```

b. Operator "break"

The break operator is used for prematurely exiting the loop, before it has completed its execution in a natural way. When the loop reaches the break operator it is terminated and the program's execution continues from the line immediately after the loop's body. A loop's termination with the break operator can only be done from its body, during an iteration of the loop. When break is executed the code in the loop's body after it is skipped and not executed. We will demonstrate exiting from loop with break with an example.

c. Calculating Factorial – Example

In this example we will calculate the factorial of a number entered from the console. The calculation is performed by using an infinite while loop and the operator break. Let's remember from the mathematics what is factorial and how it is calculated. The factorial of an integer n is a function that is calculated as a product of all integers less than or equal to n or equal to it. It is written down as n! and by definition the following formulas are valid for it: - N! = 1 * 2 * 3 … (n-1) * n, for n> 1; - 2! = 1 * 2; - 1! = 1; - 0! = 1. The product n! can be expressed by a factorial of integers less than n: - N! = (N-1)! * N, by using the initial value of 0! = 1. In order to calculate the factorial of n we will directly use the definition:

```
int n = int.Parse(Console.ReadLine());
// "decimal" is the biggest C# type that can hold integer values
decimal factorial = 1;
 // Perform an "infinite loop"
 while (true) {
if (n <= 1)  {
  break;
 }
  factorial *= n;  n--;
}
 Console.WriteLine("n! = " + factorial);
```

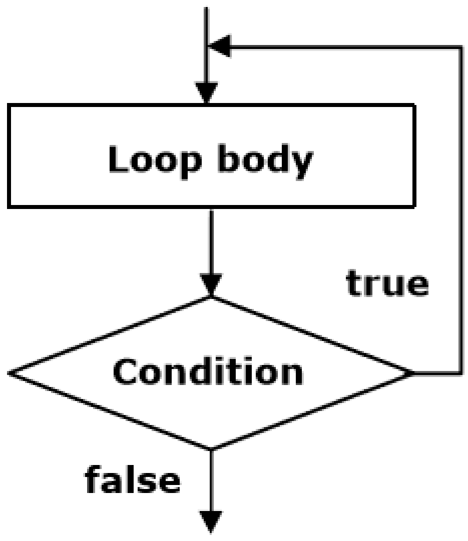
First we initialize the variable factorial with 1 and read n from the console. We construct an endless while loop by using true as a condition of the loop. We use the break operator, in order to terminate the loop, when n reaches a value less than or equal to 1. Otherwise, we multiply the current result by n and we reduce n with one unit. Practically in the first iteration of the loop the variable factorial has a value n, in the second – n*(n-1) and so on. In the last iteration of the loop the value of factorial is the product n*(n-1)*(n2)*…*3*2, which is the desired value of n!.

d. Do-While Loops

The do-while loop is similar to the while loop, but it checks the condition after each execution of its loop body. This type of loops is called loops with condition at the end (post-test loop). A do-while loop looks like this:

```
do
{
   executable code;
} while (condition);
```

By design do-while loops are executed according to the following scheme:



Initially the loop body is executed. Then its condition is checked. If it is true, the loop's body is repeated, otherwise the loop ends. This logic is repeated until the condition of the loop is broken. The body of the loop is executed at least once. If the loop's condition is constantly true, the loop never ends.

The do-while loop is used when we want to guarantee that the sequence of operations in it will be executed repeatedly and at least once in the beginning of the loop.

In this example we will again calculate the factorial of a given number n, but this time instead of an infinite while loop we will use a do-while. The logic is similar to that in the previous example:

```
Console.Write("n = ");
int n = int.Parse(Console.ReadLine());
decimal factorial = 1;
do
{
  factorial *= n;
  n--;
} while (n > 0);
Console.WriteLine("n! = " + factorial);
```

At the beginning we start with a result of 1 and multiply consecutively the result at each iteration by n, and reduce n by one unit, until n reaches 0. This gives us the product n*(n-1)*…*1. Finally, we print the result on the console. This algorithm always performs at least one multiplication and that's why it will not work properly when n ≤ 0. Here is the result of the above example's execution for n=7:

```
n = 7
n! = 5040
```

Time Boxing

Activity Name	Activity Time	Total Time
Login Systems + Setting up Visual Studio Environment	3 mints + 5 mints	8 mints
Walk through Theory & Tasks	60 mints	60 mints
Implement Tasks	80 mints	80 mints
Evaluation Time	30 mints	30 mints
	Total Duration	178 mints

Objectives/Outcomes

This exercise delivers the idea/ concept of:

- Creation and implementation of while loop

Lab Tasks/Practical Work

1. Fibonacci series (0,1,1,2,3,5,8…) for and while loop
2. Repeatedly print the value of the variable xValue, decreasing it by 0.5 each time, as long as xValue remains positive. (while loop)
3. Print the square roots of the first 25 odd positive integers. (while loop)

Lab Manual for Computer Programming

Lab No. 8

Implementation of ARRAYS

Objectives

To understand basic concept, working and Implementation of Array.

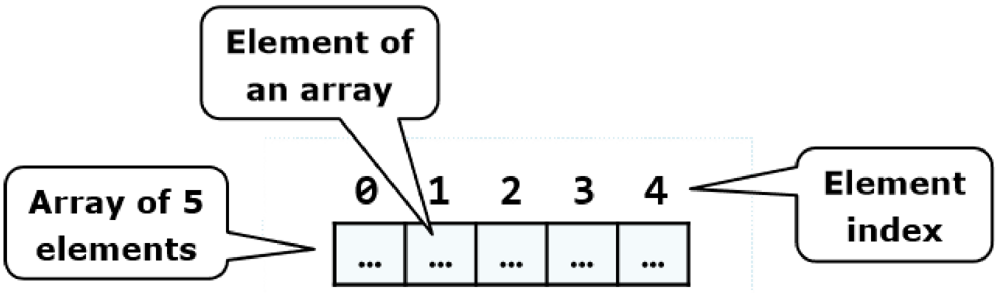
LAB # 08

Implementation of Arrays

Introduction

In this chapter we will learn about arrays as a way to work with sequences of elements of the same type. We will explain what arrays are, how we declare, create, instantiate and use them. We will examine one-dimensional arrays.

Arrays are vital for most programming languages. They are collections of variables, which we call elements:



An array's elements in C# are numbered with 0, 1, 2, … N-1. Those numbers are called indices. The total number of elements in a given array we call length of an array. All elements of a given array are of the same type, no matter whether they are primitive or reference types. This allows us to represent a group of similar elements as an ordered sequence and work on them as a whole. Arrays can be in different dimensions, but the most used are the onedimensional and the two-dimensional arrays. One-dimensional arrays are also called vectors and two-dimensional are also known as matrices.

In C# the arrays have fixed length, which is set at the time of their instantiation and determines the total number of elements. Once the length of an array is set we cannot change it anymore.

Declaring an Array

We declare an array in C# in the following way:

```
int[] myArray;
```

Creation of an Array – the Operator "new"

In C# we create an array with the help of the keyword new, which is used to allocate memory:

```
int[] myArray = new int[6];
```

Array Initialization and Default Values

Before we can use an element of a given array, it has to be initialized or to have a default value. We can do this in different ways. Here is one of them:

```
int[] myArray = { 1, 2, 3, 4, 5, 6 };
```

Here is one more example how to declare and initialize an array:

```
string[] daysOfWeek =
  { "Monday", "Tuesday", "Wednesday","Thursday", "Friday",
  "Saturday", "Sunday" };
```

Boundaries of an Array

Arrays are by default zero-based, which means the enumeration of the elements starts from 0. The first element has the index 0, the second – 1, etc. In an array of N elements, the last element has the index N-1.

Access to the Elements of an Array

We access the array elements directly using their indices. Each element can be accessed through the name of the array and the element's index (consecutive number) placed in the brackets. We can access given elements of the array both for reading and for writing, which means we can treat elements as variables. Here is an example for accessing an element of an array:

```
myArray[index] = 100;
```

Here is an example, where we allocate an array of numbers and then we change some of them:

```
int[] myArray = new int[6];
myArray[1] = 1;
myArray[5] = 5;
```

We can iterate through the array using a loop statement. The most common form of such iteration is by using a for-loop:

```
int[] arr = new int[5];
for (int i = 0; i < arr.Length; i++)
{
   arr[i] = i;
}
```

Reading an Array from the Console

Initially we read a line from the console using Console.ReadLine(), and then we parse that line to an integer number using int.Parse() and we set it to the variable n. We then use the number n as length of the array.

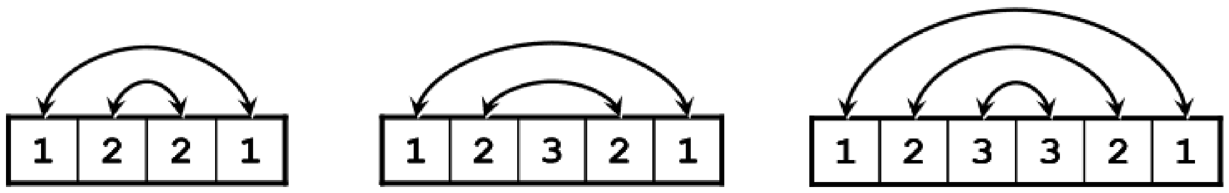
```
int n = int.Parse(Console.ReadLine());
int[] array = new int[n];
```

Again we use a loop to iterate through the array. At each iteration we set the current element to what we have read from the console. The loop will continue n times, which means it will iterate through the array and so we will read a value for each element of the array:

```
for (int i = 0; i < n; i++)
{
   array[i] = int.Parse(Console.ReadLine());
}
```

Check for Symmetric Array – Example

An array is symmetric if the first and the last elements are equal and at the same time the second element and the last but one are equal as well and so on. On the figure a few examples for symmetric arrays are shown:



In the next example we will check whether an array is symmetric:

```
Console.Write("Enter a positive integer: ");
int n = int.Parse(Console.ReadLine());
int[] array = new int[n];

Console.WriteLine("Enter the values of the array:");

for (int i = 0; i < n; i++)
{
   array[i] = int.Parse(Console.ReadLine());
}

bool symmetric = true;
for (int i = 0; i < array.Length / 2; i++)
{
   if (array[i] != array[n - i - 1])
   {
      symmetric = false;
      break;
   }
}

Console.WriteLine("Is symmetric? {0}", symmetric);
```

We print the elements of an array by hand, by using a for-loop:

```
string[] array = { "one", "two", "three", "four" };

for (int index = 0; index < array.Length; index++)
{
   // Print each element on a separate line
   Console.WriteLine("Element[{0}] = {1}", index, array[index]);
}
```

Time Boxing

Activity Name	Activity Time	Total Time
Login Systems + Setting up Visual Studio Environment	3 mints + 5 mints	8 mints
Walk through Theory & Tasks	60 mints	60 mints
Implement Tasks	80 mints	80 mints
Evaluation Time	30 mints	30 mints
	Total Duration	178 mints

Objectives/Outcomes

This exercise delivers the idea/ concept of:

- Limitation and Advantages of arrays.
- Implementation of arrays.

Lab Tasks/Practical Work

1. Write a program, which creates an array of 20 elements of type integer and initializes each of the elements with a value equals to the index of the element multiplied by 5. Print the elements to the console.
   Array[2]=2*5

2. Write a program, which reads two arrays from the console and checks whether they are equal (two arrays are equal when they are of equal length and all of their elements, which have the same index, are equal).

Array[6]={1,1,1,2,2,9}

Array2[6]={1,1,1,2,2,2}

3. Make a program in C# in which take 5 numbers from user and then give sum and avg. of them. Using arrays.

Lab Manual for Computer Programming

Lab No. 9

Implementation of 2-D ARRAY

Objectives

To understand basic concept, working, usage and implementation of 2-D array.

LAB # 09

Introduction to 2-D Array

Introduction

In this lab we will learn about multi-dimensional. The one-dimensional arrays are known also as vectors in mathematics. Often we need arrays with more than one dimension. For example, we can easily represent the standard chess board as a two-dimensional array with size 8 by 8 (8 cells in a horizontal direction and 8 cells in a vertical direction). We declare a two-dimensional with int[,].

```
int[,] twoDimensionalArray;
```

Those arrays we will call two-dimensional, because they have two dimensions. They are also known as matrices (it is mathematical term). In general arrays with more than one dimension we will call multidimensional. This way we can declare three-dimensional arrays as we add one more dimension:

```
int[,,] threeDimensionalArray;
```

In theory there is no limit for an array dimensions, but in practice we do not use much arrays with more than two dimensions therefore we will focus on two-dimensional arrays.

Multidimensional Array Declaration and Allocation We declare multidimensional arrays in a way similar to one-dimensional arrays. Each dimension except the first is marked with comma:

```
int[,] intMatrix;
float[,] floatMatrix;
string[,,] strCube;
```

In the example above we create two-dimensional and three-dimensional arrays. Each dimension is represented by a comma in the square brackets []. We are allocating memory for multidimensional arrays by using the keyword new and for each dimension we set a length in the brackets as shown:

```
int[,] intMatrix = new int[3, 4];
float[,] floatMatrix = new float[8, 2];
string[,,] stringCube = new string[5, 5, 5];
```

In this example intMatrix is a two-dimensional array with 3 elements of type int[] and each of those 3 elements has a length of 4. Two-dimensional arrays are difficult to understand explained that way. Therefore we can imagine them as two-dimensional matrices, which have rows and columns for the dimensions:

	0	1	2	3
0	1	3	6	2
1	8	5	9	1
2	4	7	3	0

The rows and the columns of the square matrices are numbered with indices from 0 to n-1. If a two-dimensional array has a size of m by n, there are exactly m*n elements.

Two-Dimensional Array Initialization

We initialize two-dimensional arrays in the same way as we initialize one-dimensional arrays. We can list the element values straight after the declaration:

```
int[,] matrix =
{
  {1, 2, 3, 4}, // row 0 values
  {5, 6, 7, 8}, // row 1 values
};
// The matrix size is 2 x 4 (2 rows, 4 cols)
```

In the example above, we initialize a two-dimensional array of type integer with size of 2 rows and 4 columns. In the outer brackets we place the elements of the first dimension, i.e. the rows of the array. Each row contains one dimensional array, which we know how to initialize.

Accessing the Elements of a Multidimensional Array

Matrices have two dimensions and respectively we access each element by using two indices: one for the rows and one for the columns. Multidimensional arrays have different indices for each dimension.

The above array matrix has 8 elements, stored in 2 rows and 4 columns. Each element can be accessed in the following way:

```
matrix[0, 0]  matrix[0, 1]  matrix[0, 2]  matrix[0, 3]
matrix[1, 0]  matrix[1, 1]  matrix[1, 2]  matrix[1, 3]
```

In this example we can access each element using indices. If we assign the index for rows to row, and the index for columns to col, then we can access any element as shown:

```
matrix[row, col]
```

Length of Multidimensional Arrays

Each dimension of a multidimensional array has its own length, which can be accessed during the execution of the program.

We can get the number of the rows of this two-dimensional array by using matrix.GetLength(0) and the number of all columns per row with matrix.GetLength(1). So, in this case matrix.GetLength(0) returns 2 and matrix.GetLength(1) returns 4.

Printing Matrices – Example

In the next example we will demonstrate how we can print two-dimensional arrays to the console:

```
// Declare and initialize a matrix of size 2 x 4
int[,] matrix =
{
  {1, 2, 3, 4}, // row 0 values
  {5, 6, 7, 8}, // row 1 value
};

// Print the matrix on the console
for (int row = 0; row < matrix.GetLength(0); row++)
{
  for (int col = 0; col < matrix.GetLength(1); col++)
  {
    Console.Write(matrix[row, col]);
  }
  Console.WriteLine();
}
```

First we declare and initialize an array, which we want to iterate through and print to the console. The array is two-dimensional, therefore we use a for loop which will iterate through the rows and a nested for loop which for each row will iterate through the columns. At each iteration we will print the current element using the appropriate method to access this element by using its two indices (row and column).

Reading Matrices from the Console – Example

In this example we will learn how to read a two-dimensional array from the console. First, we read the values (lengths) of the two-dimensions and then by using two nested loops we assign the value of each element (and in the end we print out the values of the array):

```
Console.Write("Enter the number of the rows: ");
int rows = int.Parse(Console.ReadLine());

Console.Write("Enter the number of the columns: ");
int cols = int.Parse(Console.ReadLine());

int[,] matrix = new int[rows, cols];

Console.WriteLine("Enter the cells of the matrix:");

for (int row = 0; row < rows; row++)
{
   for (int col = 0; col < cols; col++)
   {
      Console.Write("matrix[{0},{1}] = ",row, col);
      matrix[row, col] = int.Parse(Console.ReadLine());
   }
}

for (int row = 0; row < matrix.GetLength(0); row++)
{
   for (int col = 0; col < matrix.GetLength(1); col++)
   {
      Console.Write(" " + matrix[row, col]);
   }
   Console.WriteLine();
}
```

The program output when we execute it (in this case the array consists of three rows and two columns) is:

```
Enter the number of the rows: 3
Enter the number of the columns: 2
Enter the cells of the matrix:
matrix[0,0] = 2
matrix[0,1] = 3
matrix[1,0] = 5
matrix[1,1] = 10
matrix[2,0] = 8
matrix[2,1] = 9
 2 3
 5 10
 8 9
```

Time Boxing

Activity Name	Activity Time	Total Time
Login Systems + Setting up Visual Studio Environment	3 mints + 5 mints	8 mints
Walk through Theory & Tasks	60 mints	60 mints
Implement Tasks	80 mints	80 mints
Evaluation Time	30 mints	30 mints
	Total Duration	178 mints

Objectives/Outcomes

This lab exercise delivers the idea/ concept of:

- Limitation and Advantages of 2-D array
- Implementation & usage of 2-D array.

Lab Tasks/Practical Work

1) Enter the first matrix and then display it. Secondly, enter the second matrix and then display it. In the result by apply 2D arrays show multiplication of 2 matrixes.
2) Take N number of user data input and make sure N is greater then 10, which contain name of the user, his/her nationality and his/her eye color. You have to show the max color of eye in each country.
3) Write a program which by given N numbers, K and S, finds K elements out of the N numbers, the sum of which is exactly S or says it is not possible. Example:
   {3, 1, 2, 4, 9, 6}, S = 14, K = 3 → yes (1 + 2 + 4 = 14)

4) Make a program in C# in which take no. of items, price of items, quantity of items and name of items as input from the user and give the discount according to the following conditions (Use 2D Array):
   a. If from Bahria University give discount of 30%.
   b. Else if the total amount is greater than 50,000 and less than 100,000 give discount of 20%.
   c. Else if the total amount is greater than 100,000 give discount of 30%.

Lab Manual for Computer Programming

Lab No. 10

Implementation Methods

Objectives

To understand basic concept, working and usage of methods.

LAB # 10

Implementation of Methods

Introduction

In this lab we will get more familiar with what methods are and why we need to use them. The reader will be shown how to declare methods, what parameters are and what a method's signature is, how to call a method, how to pass arguments of methods and how methods return values. At the end of this chapter we will know how to create our own method and how to use (invoke) it whenever necessary. Eventually, we will suggest some good practices in working with methods.

A method is a basic part of a program. It can solve a certain problem, eventually take parameters and return a result. A method represents all data conversion a program does, to resolve a particular task. Methods consist of the program's logic. Moreover, they are the place where the "real job" is done. That is why methods can be taken as a base unit for the whole program. This on the other hand, gives us the opportunity, by using a simple block, to build bigger programs, which resolve more complex and sophisticated problems. Below is a simple example of a method that calculates rectangle's area:

```
static double GetRectangleArea(double width, double height)
{
   double area = width * height;
   return area;
}
```

Method Declaration

To declare a method means to register the method in our program. This is shown with the following declaration:

```
[static] <return_type> <method_name>([<param_list>])
```

As can be seen the type of returned value is void (i.e. that method does not return a result), the method's name is Main, followed by round brackets, between which is a list with the method's parameters. In the particular example it is actually only one parameter – the array string[] args. The sequence, in which the elements of a method are written, is strictly defined. Always, at the very first place, is the type of the value that method returns <return_type>, followed by the method's name <method_name> and list of parameters at the end <param_list> placed between in round brackets – "(" and ")". Optionally the declarations can have access modifiers (as public and static).

The list with parameters is allowed to be void (empty). In that case the only thing we have to do is to type "()" after the method's name. Although the method has not parameters the round brackets must follow its name in the declaration.

For now we will not focus at what <return_type> is. For now we will use void, which means the method will not return anything. Later, we will see how that can be changed The keyword static in the description of the declaration above is not mandatory but should be used in small simple programs. It has a special purpose that will be explained later in this chapter. Now the methods that we will use for example, will include the keyword static in their declaration.

Rules to Name a Method

It is recommended, when declare a method, to follow the rules for method naming suggested by Microsoft: - The name of a method must start with capital letter. - The PascalCase rule must be applied, i.e. each new word, that concatenates so to form the method name, must start with capital letter. - It is recommended that the method name must consist of verb, or verb and noun. Note that these rules are not mandatory, but recommendable. If we aim our C# code to follow the style of all good programmers over the globe, we must use Microsoft's code convention. A more detailed recommendation about method naming will be given in the chapter "High-Quality Code", section "Naming Methods". Here some examples for well named methods:

```
Print
GetName
PlayMusic
SetUserName
```

And some examples for bad named methods:

```
Abc11
Yellow___Black
foo
_Bar
```

Method Overloading

When in a class a method is declared and its name coincides with the name of another method, but their signatures differ by their parameters list (count of the method's parameters or the way they are arranged), we say that there are different variations / overloads of that method (method overloading). As an example, let's assume that we have to write a program that draws letters and digits to the screen. We also can assume that our program has methods for drawing strings DrawString(string str), integers – DrawInt(int number), and floating point digits – DrawFloat(float number) and so on:

```
static void DrawString(string str)
{
   // Draw string
}

static void DrawInt(int number)
{
   // Draw integer
}

static void DrawFloat(float number)
{
   // Draw float number
}
```

Time Boxing

Activity Name	Activity Time	Total Time
Login Systems + Setting up Visual Studio Environment	3 mints + 5 mints	8 mints
Walk through Theory & Tasks	60 mints	60 mints
Implement Tasks	80 mints	80 mints
Evaluation Time	30 mints	30 mints
	Total Duration	178 mints

Objectives/Outcomes

This lab exercise delivers the idea/ concept of:

- Working of Methods
- Limitation and Advantages of methods

Lab Tasks/Practical Work

1. Write a method named square_cube() that computes the square and cube of the value passed to it and display the result. Ask the user to provide the integer input in the main() and then call the function.
2. Write a method table() which generates multiplicative table of an integer. The function receives three integers as its arguments. The first argument determine the table to be generated while the second and the third integer tell the starting and ending point respectively. Ask the user to provide the three integer as input in the main().
3. Write a C# program that inputs a character array and check if the odd positions in an array contains vowels or not? Make two methods one for array input second for vowel checking.
4. Take input of an array in on method and print reverse of that array using second method.

Lab Manual for Computer Programming

Lab No. 11

Filing

Objectives

To understand basic concept, working and usage of exception handling.

LAB # 12

Filing

C# stream

Stream provides a generic interface to the types of input and output, and isolate the programmer from the specific details of the operating system and the underlying devices. For instance, MemoryStream works with data located in the memory and FileStream with data in a files.

C# read text file with File.ReadAllText

The `File.ReadAllText()` method opens a text file, reads all lines of the file into a string, and then closes the file.

Example 1

```
using System;
using System.IO;
using System.Text;

namespace ReadAllText
{
    class Program
    {
        static void Main(string[] args)
        {
            var path = @"C:\Users\Jano\Documents\thermopylae.txt";

            string content = File.ReadAllText(path, Encoding.UTF8);
            Console.WriteLine(content);
        }
    }
}
```

C# read text file with File.ReadAllLines

The `File.ReadAllLines()` opens a text file, reads all lines of the file into a string array, and then closes the file.

The `FileStream` class provides a `Stream` for a file, supporting both synchronous and asynchronous read and write operations. The constructor initializes a new instance of the `FileStream` class with the specified path, creation mode, and read/write permission.

Example 2

```
using System;
using System.IO;
using System.Text;

namespace ReadAllLines
{
    class Program
    {
        static void Main(string[] args)
        {
            var path = @"C:\Users\Jano\Documents\thermopylae.txt";

            string[] lines = File.ReadAllLines(path, Encoding.UTF8);

            foreach (string line in lines)
            {
                Console.WriteLine(line);
            }
        }
    }
}
```

C# reading text file with StreamReader

`StreamReader` is designed for character input in a particular encoding. It is used for reading lines of information from a standard text file.

Using StreamReader's ReadToEnd

The `ReadToEnd()` method reads all characters from the current position of the stream to its end.

Example 3

```
using System;
using System.IO;
using System.Text;

namespace StreamReaderReadToEnd
{
    class Program
    {
        static void Main(string[] args)
        {
            var path = @"C:\Users\Jano\Documents\thermopylae.txt";

            using var fs = new FileStream(path, FileMode.Open, FileAccess.Read);
            using var sr = new StreamReader(fs, Encoding.UTF8);

            string content = sr.ReadToEnd();

            Console.WriteLine(content);
        }
    }
}
```

`StreamReader's ReadToEnd()` method reads all characters from the current position to the end of the file.

Using StreamReader's ReadLine

The `ReadLine()` method of the `StreamReader` reads a line of characters from the current stream and returns the data as a string.

Example 4

```
using System;
using System.IO;
using System.Text;

namespace StreamReaderReadLine
{
    class Program
    {
        static void Main(string[] args)
        {
            var path = @"C:\Users\Jano\Documents\thermopylae.txt";

            using var fs = new FileStream(path, FileMode.Open, FileAccess.Read);
            using var sr = new StreamReader(fs, Encoding.UTF8);

            string line = String.Empty;

            while ((line = sr.ReadLine()) != null)
            {
                Console.WriteLine(line);
            }
        }
    }
}
```

Example 5

The code example reads a file line by line.

```
string line = String.Empty;
while ((line = streamReader.ReadLine()) != null)
{
        Console.WriteLine(line);
}
```

In a while loop, we read the contents of the file line by line with the `StreamReader's ReadLine()` method.

Time Boxing

Activity Name	Activity Time	Total Time
Login Systems + Setting up Visual Studio Environment	3 mints + 5 mints	8 mints
Walk through Theory & Tasks	60 mints	60 mints
Implement Tasks	80 mints	80 mints
Evaluation Time	30 mints	30 mints
	Total Duration	178 mints

Objectives/Outcomes

This Lab exercise delivers the idea/concept of:

- Creation and implementation of filing

Lab Tasks/Practical Work

1. Design a program of Employee in which you have to take information of 05 employees. Information includes (employee_id, name, date of birth, email, residential address, job title, salary…etc.) and save all the records in a txt file using StreamWriter.
2. Design a program of Grocery items in which you have to take data of 15 items. Items includes (item_id, item name, date of manufacturing, date of expiration, quantity, price…etc.). Save all the data in a txt file using StreamWriter and print the data using StreamReader.

Lab Manual for Computer Programming

Lab No. 12

Implementing Exception Handling

Objectives

To understand basic concept, working and usage of exception handling.

LAB # 12

Implementing Exception Handling

Introduction

In this lab we will learn how to handle exceptions using the try-catch construct, how to pass them to the calling methods and how to throw standard or our own exceptions using the throw construct. We will give various examples for using exceptions. We will look at the types of exceptions and the exceptions hierarchy in the .NET Framework. At the end, we will look at the advantages of using exceptions, best practices and how to apply them in different situations.

What are Exceptions

Exceptions are a type of error that occurs during the execution of an application. Errors are typically problems that are not expected. Whereas, exceptions are expected to happen within the application's code for various reasons.

Applications use exception handling logic to explicitly handle the exceptions when they happen. Exceptions can occur for a wide variety of reasons. From the infamous NullReferenceException to a database query timeout.

The Anatomy of C# Exceptions

Exceptions allow an application to transfer control from one part of the code to another. When an exception is thrown, the current flow of the code is interrupted and handed back to a parent try catch block. C# exception handling is done with the follow keywords: try, catch, finally, and throw

- **try** – A try block is used to encapsulate a region of code. If any code throws an exception within that try block, the exception will be handled by the corresponding catch.
- **catch** – When an exception occurs, the Catch block of code is executed. This is where you are able to handle the exception, log it, or ignore it.
- **finally** – The finally block allows you to execute certain code if an exception is thrown or not. For example, disposing of an object that must be disposed of.
- **throw** – The throw keyword is used to actually create a new exception that is the bubbled up to a try catch finally block.

Your exception handling code can utilize **multiple C# catch statements for different types of exceptions**.

Common .NET Exceptions

Proper exception handling is critical to all application code. There are a lot of standard exceptions that are frequently used. The most common being the dreaded null reference exception. These are some of the common C# Exception types that you will see on a regular basis.

The follow is a list of common .NET exceptions:

- **System.NullReferenceException** – Very common exception related to calling a method on a null object reference
- **System.IndexOutOfRangeException** – Occurs attempting to access an array element that does not exist
- **System.IO.IOException** – Used around many file I/O type operations
- **System.Net.WebException** – Commonly thrown around any errors performing HTTP calls
- **System.Data.SqlClient.SqlException** – Various types of SQL Server exceptions
- **System.StackOverflowException** – If a method calls itself recursively, you may get this exception
- **System.OutOfMemoryException** – If your app runs out of memory
- **System.InvalidCastException** – If you try to cast an object to a type that it can't be cast to
- **System.InvalidOperationException** – Common generic exception in a lot of libraries
- **System.ObjectDisposedException** – Trying to use an object that has already been

Sample File Code 1:

```
using System;

using System.IO;

using System.Text;

public class Writer

{

  static string ans="y";

  public static void Main(String[] args)

  {

     Writing();

  }

  static void Writing()

  {

     if (ans=="y" || ans=="Y")

     {

          Console.Write ("Enter the file name: ");

          string Filename = Console.ReadLine();

if (!File.Exists(Filename))
{
  Console.WriteLine("{0} does not exist!",Filename);
  return;
}
 StreamWriter sr = File.AppendText(Filename);
Console.Write ("Enter a string to be written to the file: ");
 String str = Console.ReadLine();
 sr.WriteLine(str);
 sr.Close();
 Console.Write ("Do you want to continue [Y/N]: ");
 ans= Console.ReadLine();
 Writing();
       }
   }
}
```

Sample File Code 2:

```
static void Main()
    {
        // These examples assume a "C:\Users\Public\TestFolder" folder on your machine.
        // You can modify the path if necessary.


         // Example #1: Write an array of strings to a file.
        // Create a string array that consists of three lines.
        string[] lines = { "First line", "Second line", "Third line" };
        // WriteAllLines creates a file, writes a collection of strings to the file,
        // and then closes the file.  You do NOT need to call Flush() or Close().

        System.IO.File.WriteAllLines(@"C:\Users\Public\TestFolder\WriteLines.txt", lines);

 }
```

Time Boxing

Activity Name	Activity Time	Total Time
Login Systems + Setting up Visual Studio Environment	3 mints + 5 mints	8 mints
Walk through Theory & Tasks	60 mints	60 mints
Implement Tasks	80 mints	80 mints
Evaluation Time	30 mints	30 mints
	Total Duration	178 mints

Objectives/Outcomes

This Lab exercise delivers the idea/concept of:

- Creation and implementation of Exception Handling

Lab Tasks/Practical Work

3. Write a program that takes a positive integer from the console and prints the square root of this integer. If the input is negative or invalid print "Invalid Number" in the console. In all cases print "Good Bye".
4. Write a method ReadNumber(int start, int end) that reads an integer array of 10 values from the console in the range [start…end]. In case the input integer is not valid, or it is not in the required range throw appropriate exception.
5. Write a method that takes as a parameter the name of a text file then, reads the file and returns its content as string. What should the method do if an exception is thrown?

Lab Manual for Computer Programming

Lab No. 13

Implementing Recursion

Objectives

To understand Factorial, Fibonacci Series, Binomial coefficients and solving the tower of Hanoi by using recursive functionality.

LAB # 13

Implementing Recursion

Tower of Hanoi

The Tower of Hanoi or Towers of Hanoi is a mathematical game or puzzle. It consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

- Only one disk may be moved at a time
- Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- No disk may be placed on top of a smaller disk.

Fibonacci Series

It is a simple numerical series that is the foundation for an incredible mathematical relationship behind phi.

Starting with 0 and 1, each new number in the series is simply the sum of the two before it.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, . . .

The ratio of each successive pair of numbers in the series approximates phi (1.618. . .) , as 5 divided by 3 is 1.666..., and 8 divided by 5 is 1.60.

Factorial

The factorial is of a non-negative integer n, denoted by n!, is the product of all positive integers less than or equal to n.

For example,

The factorial operation is encountered in many different areas of mathematics, notably in combinatorics, algebra and mathematical analysis. Its most basic occurrence is the fact that there are n! ways to arrange n distinct objects into a sequence (i.e., permutations of a the set of objects).

Time Boxing

Activity Name	Activity Time	Total Time
Login Systems + Setting up Visual Studio Environment	3 mints + 5 mints	8 mints
Walk through Theory & Tasks	60 mints	60 mints
Implement Tasks	80 mints	80 mints
Evaluation Time	30 mints	30 mints
	Total Duration	178 mints

Objectives/Outcomes

This Lab exercise delivers the idea/concept of:

Creation and implementation of recursive methods

Lab Tasks/Practical Work

1. Write a code which prints the following series:

   2 4 8 16 - - - n

2. Write a program to calculate factorial of any given number using recursion.
3. Write a program to print Fibonacci series using recursion.

Lab Manual for Computer Programming

Lab No. 11

Structures and Pointers

Objectives

To understand basic concept, working and usage of exception handling.

LAB # 14

Structures and Pointers C# stream

In C#, a structure is a value type data type. It helps you to make a single variable hold related data of various data types. The **struct** keyword is used for creating a structure.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

- Title
- Author
- Subject
- Book ID

Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member for your program.

For example, here is the way you can declare the Book structure −

```
struct Books {
   public string title;
   public string author;
   public string subject;
   public int book_id;
};
```

The following program shows the use of the structure −

```
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";
      Book1.subject = "C Programming Tutorial";
      Book1.book_id = 6495407;

      /* book 2 specification */
      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();
   }
}
```

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

```
Book 1 title : C Programming

Book 1 author : Nuha Ali

Book 1 subject : C Programming Tutorial

Book 1 book_id : 6495407

Book 2 title : Telecom Billing

Book 2 author : Zara Ali

Book 2 subject : Telecom Billing Tutorial

Book 2 book_id : 6495700
```

Features of C# Structures

You have already used a simple structure named Books. Structures in C# are quite different from that in traditional C or C++. The C# structures have the following features −

- Structures can have methods, fields, indexers, properties, operator methods, and events.
- Structures can have defined constructors, but not destructors. However, you cannot define a default constructor for a structure. The default constructor is automatically defined and cannot be changed.
- Unlike classes, structures cannot inherit other structures or classes.
- Structures cannot be used as a base for other structures or classes.
- A structure can implement one or more interfaces.
- Structure members cannot be specified as abstract, virtual, or protected.
- When you create a struct object using the **New** operator, it gets created and the appropriate constructor is called. Unlike classes, structs can be instantiated without using the New operator.
- If the New operator is not used, the fields remain unassigned and the object cannot be used until all the fields are initialized.

C# supports pointers in a limited extent. A C# pointer is nothing but a variable that holds the memory address of another type. But in C# pointer can only be declared to hold the memory address of value types and arrays. Unlike reference types, pointer types are not tracked by the default garbage collection mechanism. For the same reason pointers are not allowed to point to a reference type or even to a structure type which contains a reference type. We can say that pointers can point to only unmanaged types which includes all basic data types, enum types, other pointer types and structs which contain only unmanaged types.

Declaring a Pointer type

The general form of declaring a pointer type is as shown below,

*type *variable_name;*

Where * is known as the de-reference operator. For example the following statement

*int *x ;*

Declares a pointer variable x, which can hold the address of an int type. The reference operator (&) can be used to get the memory address of a variable.

```
int x = 100;
```

The &x gives the memory address of the variable x, which we can assign to a pointer variable

```
int *ptr = & x;.
Console.WriteLine((int)ptr) // Displays the memory address
Console.WriteLine(*ptr) // Displays the value at the memory addres
s.
```

Unsafe Codes

The C# statements can be executed either as in a safe or in an unsafe context. The statements marked as unsafe by using the keyword unsafe runs outside the control of Garbage Collector. Remember that in C# any code involving pointers requires an unsafe context.

We can use the unsafe keyword in two different ways. It can be used as a modifier to a method, property, and constructor etc. For example

```
// Author: rajeshvs@msn.com
using System;
class MyClass
{
    public unsafe void Method()
    {
        int x = 10;
        int y = 20;
        int* ptr1 = &x;
            int* ptr2 = &y;
            Console.WriteLine((int)ptr1);
            Console.WriteLine((int)ptr2);
            Console.WriteLine(*ptr1);
```

```
                Console.WriteLine(*ptr2);
            }
        }
        class MyClient
        {
            public static void Main()
            {
                MyClass mc = new MyClass();
                mc.Method();
            }
        }
```

The keyword unsafe can also be used to mark a group of statements as unsafe as shown below.

```
// Author: rajeshvs@msn.com
//unsafe blocks
using System;
class MyClass
{
    public void Method()
    {
        unsafe
        {
                int x = 10;
                int y = 20;
                int* ptr1 = &x;
                int* ptr2 = &y;
                Console.WriteLine((int)ptr1);
                Console.WriteLine((int)ptr2);
                Console.WriteLine(*ptr1);
                Console.WriteLine(*ptr2);
            }
        }
    }
    class MyClient
    {
        public static void Main()
        {
            MyClass mc = new MyClass();
            mc.Method();
        }
    }
```

Pointers & Structures

The structures in C# are value types. The pointers can be used with structures if it contains only value types as its members. For example

```
// Author: rajeshvs@msn.com
using System;
struct MyStruct
{
    public int x;
    public int y;
    public void SetXY(int i, int j)
    {
        x = i;
        y = j;
    }
    public void ShowXY()
    {
        Console.WriteLine(x);
        Console.WriteLine(y);
    }
}
class MyClient
{
    public unsafe static void Main()
    {
        MyStruct ms = new MyStruct();
        MyStruct* ms1 = &ms;
        ms1->SetXY(10, 20);
        ms1->ShowXY();
    }
}
```

Time Boxing

Activity Name	Activity Time	Total Time
Login Systems + Setting up Visual Studio Environment	3 mints + 5 mints	8 mints
Walk through Theory & Tasks	60 mints	60 mints
Implement Tasks	80 mints	80 mints
Evaluation Time	30 mints	30 mints
	Total Duration	178 mints

Objectives/Outcomes

This Lab exercise delivers the idea/concept of:

- Creation and implementation of Structures and Pointers

Lab Tasks/Practical Work

1. Design a program of Employee in which you have to take information of 05 employees. Information includes (employee_id, name, date of birth, email, residential address, job title, salary…etc.) and save all the records using structures.