UNIT DIT503: C#.NET PROGRAMMING

UNIT CODE: L/617/6742

C# Programming Fundamentals

Introduction to C# Programming

- C# is a modern, object-oriented programming language.
- Developed by Microsoft as part of the .NET framework.
- Used for desktop applications, web development, game development (Unity), and more.
- Features:
  Type-safe
  Object-oriented
  Scalable & Maintainable
  Rich Library Support
- C# is the primary language used in ASP.NET for web development.

Structure of a C# Program

```
// Importing necessary namespaces
using System;

// Namespace declaration
namespace MyFirstApp
{
    // Class definition
    class Program
    {
        // Main method - entry point of the program
        static void Main()
        {
            // Displaying output on the console
            Console.WriteLine("Hello, C#!");
        }
    }
}
```

Comments in C#

- Comments are non-executable lines in code that provide explanations. Used for code documentation and readability.
- Types of Comments in C#:

  Single-line comments

```
// This is a single-line comment
Console.WriteLine("Hello, World!"); // This prints "Hello, World!" to the console
```

  Multi-line comments

```
/* This is a multi-line comment
   explaining the following code */
Console.WriteLine("C# Multi-line Comment Example");
```

Comments in C#

XML Documentation comments

```
/// <summary>
/// Adds two numbers and returns the sum.
/// </summary>
/// <param name="a">First number</param>
/// <param name="b">Second number</param>
/// <returns>Sum of a and b</returns>
public int Add(int a, int b)
{
    return a + b;
}
```

Data Types

- **Integer Types**

```
byte myByte = 255;          // 0 to 255
short myShort = -32768;     // -32,768 to 32,767
int myInt = 2147483647;     // -2,147,483,648 to 2,147,483,647
long myLong = 92233720368547758O7L; // -9.2e18 to 9.2e18

Console.WriteLine($"Byte: {myByte}, Short: {myShort}, Int: {myInt}, Long: {myLong}");
```

- **Floating-Point Types**

```
float myFloat = 3.14f;     // Requires 'f' at the end
double myDouble = 3.14159265358979; // More precision than float
decimal myDecimal = 3.141592653589793238m; // Highest precision (suitable for financial calcu

Console.WriteLine($"Float: {myFloat}, Double: {myDouble}, Decimal: {myDecimal}");
```

Data Types

- **Character and String Types**

```
char myChar = 'A';
string myString = "Hello, C#!";

Console.WriteLine($"Character: {myChar}, String: {myString}");
```

- **Boolean Type**

```
bool isCSharpFun = true;
bool isMathHard = false;

Console.WriteLine($"Is C# fun? {isCSharpFun}");
Console.WriteLine($"Is math hard? {isMathHard}");
```

Data Types

- **Date and Time**

```
DateTime today = DateTime.Now;
DateTime specificDate = new DateTime(2025, 3, 30);
TimeSpan duration = new TimeSpan(1, 2, 30); // 1 hour, 2 minutes, 30 seconds

Console.WriteLine($"Today: {today}");
Console.WriteLine($"Specific Date: {specificDate}");
Console.WriteLine($"Duration: {duration}");
```

- **Object Type**

```
object myObject = "I can store anything!";
Console.WriteLine($"Object value: {myObject}");
```

Printing in C#

- **Printing Simple Text**

```
Console.WriteLine("Hello, World!"); // Prints and moves to a new line
Console.Write("This is C# programming."); // Prints without moving to a new line
Console.Write(" Let's learn together!"); // Continues on the same line
```

- **Printing Variables**

```
string name = "Alice";
int age = 25;
double price = 9.99;

Console.WriteLine("Name: " + name);
Console.WriteLine("Age: " + age);
Console.WriteLine("Price: $" + price);
```

Printing in C#

- **Using String Interpolation ($ syntax)**

```
string product = "Laptop";
int quantity = 2;
double unitPrice = 799.99;
double totalPrice = quantity * unitPrice;

Console.WriteLine($"Product: {product}");
Console.WriteLine($"Quantity: {quantity}");
Console.WriteLine($"Total Price: ${totalPrice}");
```

C# Strings and Escaped Characters

- **Using Escape Sequences**

```
string message = "Hello, World!\nWelcome to C#."; // \n for a new line
string tabbedText = "Item1\tItem2\tItem3"; // \t for a tab space
string quoteExample = "He said, \"C# is awesome!\""; // \" for double quotes
string backslashExample = "Path: C:\\Program Files\\MyApp"; // \\ for backslash

Console.WriteLine(message);
Console.WriteLine(tabbedText);
Console.WriteLine(quoteExample);
Console.WriteLine(backslashExample);
```

- **Using Verbatim Strings**

```
string filePath = @"C:\Users\Admin\Documents\Project";
string multiLineText = @"This is line one.
This is line two.
This is line three.";

Console.WriteLine(filePath);
Console.WriteLine(multiLineText);
```

Operators in C#

An operator is a symbol that instructs the compiler to perform a specific operation on variables or values. Operators manipulate data and variables in a program.

- Arithmetic operators
- Relational operators
- Logical operators
- Bitwise operators
- Assignment operators

Arithmetic operators

Operator	Description	Example(a=10, b=5)	Result
+	Addition	a + b	15
-	Subtraction	a - b	5
*	Multiplication	a * b	50
/	Division	a / b	2
%	Modulus	a % b	0

Relational operators

Operator	Description	Example(a=10, b=5)	Result
==	Equal to	a == b	false
!=	Not equal to	a != b	true
>	Greater than	a > b	true
<	Less than	a < b	false
>=	Greater than or equal to	a >= b	true
<=	Less than or equal to	a <= b	false

Logical Operators

Operator	Description	Example(a=10, b=5)	Result
&&	Logical AND	(a > 5 && b > 10)	False
||	Logical OR	(a > 5 || b > 10)	True
!	Logical NOT	! (a > 5 && b > 10)	False

Assignment Operators

Operator	Description	Example(a=10, b=5)	Result
=	Assign	a = 5	5
+=	Add and assign	a +=5	a = a + 5
-=	Subtract and assign	a -=5	a = a - 5
*=	Multiply and assign	a *=5	a = a * 5
/=	Divide and assign	a /=5	a = a / 5
%=	Modulus and assign	a %=5	a = a % 5

Increment & decrement

Operator	Description	Example(a=5)	Result
++a	Pre-Increment	b = ++a	b = 6, a = 6
a++	Post-Increment	b = a++	b = 5, a = 6
--a	Pre-Decrement	b = --a	b =4, a = 4
a++	Post-Decrement	b = a--	b = 5, a = 4

Advanced Math with System.Math

```
double myValue;
myValue = Math.Sqrt(81);        // 9.0
myValue = Math.Round(42.889, 2); // 42.89
myValue = Math.Abs(-10);        // 10
myValue = Math.Log(24.212);     // Logarithm
myValue = Math.PI;              // 3.14159...
```

Type Conversions in C#

Types of Conversions

- Widening Conversions

```
int mySmallValue = Int32.MaxValue;
long myLargeValue = mySmallValue;
```

- Narrowing Conversions

```
int count32 = 1000;
short count16 = (short)count32;
```

Type Conversions in C#

Converting Between Numbers and Strings

- Using Convert Class

```
string countString = "10";
int count = Convert.ToInt32(countString);
countString = Convert.ToString(count);
```

- Using Parse Method

```
int count = Int32.Parse("10");
```

Control Statements

Control statements in C# determine the flow of execution in a program. They help in making decisions, executing loops, and controlling the sequence of execution.

Types of Control Statements in C

- **Selection Statements:** Used to execute a block of code based on conditions.
    - Examples: if, if-else, switch
- **Iteration Statements:** Used to repeat a block of code multiple times.
    - Examples: for, foreach, while, do-while
- **Jump Statements:** Used to change the normal flow of execution.
    - Examples: break, continue, return

Selection statements

- **if-else**

```
if (condition)
{
    // Code executes if condition is true
}
else
{
    // Code executes if condition is false
}
```

- **Switch Statement**

```
switch (variable)
{
    case value1:
        // Code executes if variable == value1
        break;
    case value2:
        // Code executes if variable == value2
        break;
    default:
        // Code executes if none of the cases match
        break;
}
```

Iteration Statements

- **for Loop**

```
for (initialization; condition; update)
{
    // Code executes repeatedly while the condition is true
}
```

- **foreach Loop**

```
foreach (var item in collection)
{
    // Code executes for each element in the collection
}
```

Iteration Statements

- **while Loop**

```
while (condition)
{
    // Code executes repeatedly while the condition is true
}
```

- **do-while Loop**

```
do
{
    // Code executes at least once
} while (condition);
```

Arrays in C#

An array is a collection of elements of the same data type stored in contiguous memory locations. Arrays in C# are zero-based indexed, meaning the first element is at index 0.

- **Single-Dimensional Array**

```
int[] numbers = new int[5];  // Array of size 5
numbers[0] = 10;  // Assigning values
numbers[1] = 20;

Console.WriteLine(numbers[0]);  // Output: 10
```

- **Multi-Dimensional Array**

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

Console.WriteLine(matrix[1, 2]);  // Output: 6
```

Array Manipulation Methods

Method	Description
Length	Returns the total number of elements in all dimensions of an array.
GetLowerBound()	Returns the lower bound (starting index) of a specified dimension.
GetUpperBound()	Returns the upper bound (last index) of a specified dimension.
Clear()	Clears (resets) part or all of an array's contents to their default values (e.g., 0 for integers).
IndexOf()	Searches a one-dimensional array for the first occurrence of a specified value.
LastIndexOf()	Searches a one-dimensional array for the last occurrence of a specified value.
Sort()	Sorts a one-dimensional array. Only works with comparable data types (e.g., integers, strings).
Reverse()	Reverses the elements of a one-dimensional array.

ArraysList in C#

- A collection class in C# that belongs to the System.Collections namespace.
- Unlike arrays, it is dynamic in size.
- Can hold elements of different data types (non-generic).
- It automatically resizes itself when elements are added or removed.
- Useful for scenarios where the collection size is not known in advance or might change.

```
// Creating an ArrayList
ArrayList list = new ArrayList();
```

ArrayList Manipulation Methods

Method	Description
Add()	Adds an object to the end of the ArrayList.
Insert()	Inserts an object at a specified index in the ArrayList.
Remove()	Removes the first occurrence of a specific object from the ArrayList.
RemoveAt()	Removes the element at the specified index.
Clear()	Removes all elements from the ArrayList.
Contains()	Checks if the ArrayList contains a specified object.
IndexOf()	Returns the index of the first occurrence of the specified object in the ArrayList.
LastIndexOf()	Returns the index of the last occurrence of the specified object in the ArrayList.

ArrayList Manipulation Methods

Method	Description
InsertRange()	Inserts a collection of objects at the specified index in the ArrayList.
RemoveRange()	Removes a specified range of elements from the ArrayList.
Capacity	Gets or sets the number of elements that the ArrayList can store before resizing.
Count	Gets the number of elements contained in the ArrayList.
ToArray()	Converts the ArrayList to a regular array of objects.
Sort()	Sorts the elements in the ArrayList. The elements must be comparable (like numbers or strings).
Reverse()	Reverses the elements in the ArrayList.
TrimToSize()	Sets the capacity of the ArrayList to the actual number of elements in the list, reducing unused memory space.

Strings in C#

- A string is a sequence of characters.
- In C#, a string is an object of the System.String class.
- Strings are immutable, meaning once a string is created, it cannot be changed.

```
// Basic string
string greeting = "Hello";

// String with concatenation
string fullName = "John" + " " + "Doe";  // Result: "John Doe"

// String interpolation
string name = "John";
string message = $"Hello, {name}!";  // Result: "Hello, John!"

// Verbatim string
string filePath = @"C:\Users\John\Documents";  // Result: C:\Users\John\Documents
```

String Manipulation Methods

Method	Description
Length	Returns the number of characters in the string (as an integer).
ToUpper()	Returns a copy of the string with all the characters changed to uppercase.
ToLower()	Returns a copy of the string with all the characters changed to lowercase.
Trim()	Removes spaces (or the characters you specify) from both ends of the string.
TrimStart()	Removes spaces (or the characters you specify) from the beginning of the string.
TrimEnd()	Removes spaces (or the characters you specify) from the end of the string.
PadLeft()	Adds the specified character to the left side of a string until it reaches the specified length.
PadRight()	Adds the specified character to the right side of a string until it reaches the specified length.
Insert()	Inserts another string at a specified (zero-based) index position.

String Manipulation Methods

Method	Description
Remove()	Removes a specified number of characters from a specified position.
Replace()	Replaces a specified substring with another string.
Substring()	Extracts a portion of a string starting from the specified index and for the specified length.
StartsWith()	Determines if a string starts with a specified substring.
EndsWith()	Determines if a string ends with a specified substring.
IndexOf()	Finds the zero-based position of the first occurrence of a substring.
LastIndexOf()	Divides a string into an array of substrings based on a delimiter.
Split()	Sets the capacity of the ArrayList to the actual number of elements in the list, reducing unused memory space.
Join()	Joins an array of strings into a single string with a specified separator.

DateTime in C#

The DateTime structure is used to represent date and time values. It provides a set of properties and methods for working with dates and times in a variety of ways, including comparisons, calculations, formatting, and parsing. DateTime objects are immutable, meaning once created, their values cannot be changed. Instead, any operation that appears to modify a DateTime object will actually create a new DateTime object with the updated value.

Components of DateTime

- Date: A specific calendar date (day, month, and year).
- Time: A specific time of day, represented by hours, minutes, seconds, and milliseconds.

DateTime Formatting Specifiers

DateTime provides format strings that allow you to specify how the date/time is represented as a string. Common format specifiers include:

- "d": Short date pattern (e.g., "3/30/2025")
- "D": Long date pattern (e.g., "Sunday, March 30, 2025")
- "t": Short time pattern (e.g., "2:25 PM")
- "T": Long time pattern (e.g., "2:25:00 PM")
- "f": Full date/time pattern (e.g., "Sunday, March 30, 2025 2:25 PM")
- "F": Full date/time pattern with time (e.g., "Sunday, March 30, 2025 2:25:00 PM")
- "yyyy-MM-dd": Custom format for year, month, and day (e.g., "2025-03-30")

DateTime Manipulation Methods

Method	Description
Now	Gets the current date and time (local).
UtcNow	Gets the current date and time in UTC (Coordinated Universal Time).
Today	Gets the current date with the time set to 00:00:00.
Year, Month, Day, Hour, Minute, Second, Millisecond	Returns a specific part of the DateTime object (year, month, day, etc.).
DayOfWeek	Returns the day of the week for a DateTime (using DayOfWeek enumeration).
AddYears(), AddMonths(), AddDays(), AddHours(), AddMinutes(), AddSeconds(), AddMilliseconds()	Adds a specified number of years, months, days, hours, minutes, seconds, or milliseconds to the DateTime.

DateTime Manipulation Methods

Method	Description
Subtract()	Subtracts a TimeSpan from a DateTime.
ToString()	Converts the DateTime object to a string representation. Can specify a format string.
DaysInMonth()	Returns the number of days in a specified month and year.
IsLeapYear()	Determines if the specified year is a leap year.
Compare()	Compares two DateTime objects and returns an integer indicating their relative order.

TimeSpan in C#

A TimeSpan represents a time interval or a duration between two dates or times. It is not tied to any specific date or time but instead holds the difference between them. You can think of it as a duration that measures a period, such as a few hours, days, or even fractional milliseconds.

Common uses of TimeSpan include:

- Calculating the difference between two DateTime values.
- Representing durations (e.g., the length of an event or an elapsed time).
- Performing arithmetic operations with time intervals.

TimeSpan Manipulation Methods

Method	Description
Days, Hours, Minutes, Seconds, Milliseconds	These properties return the individual components of the TimeSpan object as integers. They represent the days, hours, minutes, seconds, or milliseconds in the current TimeSpan. For example, the Hours property will return a value between -23 and 23.
TotalDays, TotalHours, TotalMinutes, TotalSeconds, TotalMilliseconds	These properties return the total number of days, hours, minutes, seconds, or milliseconds in the TimeSpan as a double. These values can include fractional values. These totals represent the entire duration in a specific unit.
Add()	Adds a TimeSpan to the current TimeSpan and returns a new TimeSpan. You can use the + operator for the same purpose.
Subtract()	Subtracts a TimeSpan from the current TimeSpan and returns a new TimeSpan. You can also use the - operator for this operation.
FromDays(), FromHours(), FromMinutes(), FromSeconds(), FromMilliseconds()	These methods allow you to create a new TimeSpan from a specified number of days, hours, minutes, seconds, or milliseconds. This is a shorthand way of constructing a TimeSpan object.
ToString()	Converts the TimeSpan to a string representation. You can also provide a format string to specify the output format.

Methods in C#

Methods are the basic building blocks for organizing and structuring code in C#. A method is a named block of code that performs a specific task. Methods help in breaking down code into logical components and are essential for object-oriented programming (OOP).

To declare a method in C#, the syntax includes the following components:

- Return Type: Specifies the type of data the method returns (or void if no data is returned).
- Method Name: The name of the method.
- Parameters (Optional): A method can accept parameters, which are values passed to the method to perform its task.

Methods in C#

Method Accessibility:

- Private: The method is accessible only within the class.
- Public: The method is accessible from any class that has a reference to the class.

Invoking Methods:

- With no return value: MyMethodNoReturnedData();
- With a return value: The returned data can be used or ignored.
    - Example: int result = MyMethodReturnsData();
    - If the method doesn’t return anything, you cannot assign its result to a variable.

Parameters in Methods

Parameters allow methods to receive input. These inputs are provided by the caller when invoking the method. Parameters are declared inside the parentheses of the method declaration. Each parameter consists of a type and a name.

```
private int AddNumbers(int number1, int number2)
{
    return number1 + number2;
}
```

When calling a method with parameters, you provide values (arguments) for each parameter.

```
int result = AddNumbers(10, 20);  // Calling with two arguments
```

Parameters in Methods

Optional parameters allow you to define parameters that are not required when calling a method. The value of these parameters can be provided when the method is called or will default to a specified value if not provided.

```
private string GetUserName(int ID, bool useShortForm = false)
{
    // Code here
}
```

Method overloading is a powerful feature in C# that allows you to define multiple methods with the same name, but with different parameter lists. The C# compiler distinguishes the methods based on the number, types, or order of parameters passed to them.

```
private decimal GetProductPrice(int ID)
{
    // Code to get price by product ID
}

private decimal GetProductPrice(string name)
{
    // Code to get price by product name
}
```


Q
A