Questions:

Q1: Question: What is the shortcut to comment and uncomment a selected block of code in Visual Studio?

- To comment use: ctrl + k + c
- To uncomment use: Ctrl + k + u

Q2: Explain the difference between a runtime error and a logical error with examples

- 1- A **runtime error** occurs while a program is running. These errors happen due to illegal operations or the environment where the code is executed. They prevent the program from continuing its execution.

  Ex: Attempting to divide by zero, accessing a file that doesn't exist, or trying to use more memory than is available.

```
using System;

class Program
{
        static void Main()
        {
        int num1 = 10;
        int num2 = 0;
        int result = num1 / num2; // Runtime error: Division by zero
        Console.WriteLine(result);
        }
}
```

- 2-**A logical error** occurs when the program runs without crashing but produces incorrect results. These errors are due to mistakes in the program's logic or algorithm.

***Ex-*Incorrect Calculation of Average**

```
using System;

class Program
{
  static void Main()
  {
    int[] numbers = { 1, 2, 3, 4, 5 };
    int sum = 0;
    foreach (int number in numbers)
    {
```

```
            sum += number;
        }
        // Intentional logical error: dividing by numbers.Length - 1 instead of numbers.Length
        double average = (double)sum / (numbers.Length - 1);
        Console.WriteLine("Average: " + average);
    }
}
```

Q3: Why is it important to follow naming conventions such as PascalCase in C#?

- **Enhances Readability**: Makes code easier to read and understand.
- **Ensures Consistency**: Provides uniformity across the codebase.
- **Facilitates Maintenance**: Simplifies the process of updating and debugging code.
- **Supports Collaboration**: Helps multiple developers work together seamlessly.
- **Improves Documentation**: Creates self-explanatory, intuitive code.
- **Adheres to Standards**: Aligns with industry best practices and framework requirements.

Q4:

Value Types

Memory Allocation:

- Stored on the Stack: Value types are stored in stack memory, which operates on a Last In, First Out (LIFO) principle.
- Direct Storage: The actual data is stored directly in the variable.

Characteristics:

- Independent Copies: When you assign one value type to another, a copy of the value is made. Changes to one instance do not affect the other.
- Primitive Data Types: Common value types include int, double, char, and bool, as well as structures (struct)

  Ex:

```
int a = 10;

int b = a; // b is a copy of a

b = 20;

Console.WriteLine(a); // Outputs: 10

Console.WriteLine(b); // Outputs: 20
```

Reference Types

Memory Allocation:

- Stored on the Heap: Reference types are stored in heap memory, which is used for dynamic memory allocation.
- Reference Storage: The variable holds a reference (or address) to the actual data located in the heap.

Characteristics:

- Shared References: When you assign one reference type to another, both variables reference the same object in memory. Changes made through one reference are reflected in the other.
- Complex Data Types: Reference types include classes (class), arrays, delegates, and interfaces.

Ex: class Person{

```
   public string Name { get; set; }
}
```

```
Person person1 = new Person { Name = "Alice" };
Person person2 = person1; // person2 references the same object as person1
person2.Name = "Bob";
Console.WriteLine(person1.Name); // Outputs: Bob
Console.WriteLine(person2.Name); // Outputs: Bob
```

Q5: What will be the output of the following code? Explain why:

```
int a = 2, b = 7;
Console.WriteLine(a % b);
```

The Output will be: 2 (2 % 7) = 2

Q6 : Logical AND (&&)

The logical AND operator is used in conditional statements to combine multiple boolean expressions. Evaluation: It performs short-circuit evaluation, meaning if the first operand is false, the second operand is not evaluated because the result will be false regardless.

&& performs short-circuit evaluation, stopping if the first condition is false.

Example:

```
bool a = true , b = false ;

if (a && b){

   Console.WriteLine("Both are true.");

}else{

   Console.WriteLine("One or both are false.");  // This will be printed

}
```

Bitwise AND (&)

The bitwise AND operator is used to perform a bitwise AND operation on two integer operands. It compares each bit of the operands and sets the corresponding bit in the result to 1 if both bits are 1. **Evaluation**: It always evaluates both operands, regardless of the first operand's value.

& does not short-circuit; it evaluates both operands fully.

Example:

```
int x = 5;  // Binary: 0101

int y = 3;  // Binary: 0011

int result = x & y;  // Binary: 0001, which is 1

Console.WriteLine(result); // Outputs: 1
```

Q7: Question: What exception might occur if the input is invalid and how can you handle it

if the input is invalid (for example, if the user enters a non-numeric value when a number is expected), an exception that might occur is FormatException. This exception is thrown when the format of an argument is invalid, such as trying to parse a non-numeric string to an integer.

1- Invalid Input Format

Exception: FormatException, NumberFormatException, or equivalent
Cause: Input does not match the expected format (e.g., entering text where a number is expected).
Handling:

```
try { int number = int.Parse(input); }

catch (FormatException ex) {

Console.WriteLine("Invalid input format. Please enter a number."); }
```

2. Null or Empty Input

Exception: ArgumentNullException, NullPointerException, or equivalent
Cause: Input is null or empty when it should contain a value.
Handling:

```
if (string.IsNullOrEmpty(input)){
   Console.WriteLine("Input cannot be empty. Please provide valid data.");
}
```

3. Out of Range Values

Exception: OverflowException, IndexOutOfRangeException, ArgumentOutOfRangeException
Cause: Input exceeds the acceptable range or size (e.g., entering a number too large for an int).
Handling:

```
try{
   int number = Convert.ToInt32(input);
}
catch (OverflowException ex){
   Console.WriteLine("The number is too large or too small.");
}
```

Q8: Given the code below, what is the value of x after execution? Explain why

```
int x = 5;
int y = ++x + x++;
```

Ans:

X = 7

Part 02

1-Linkedin Article

2-what's the difference between compiled and interpreted languages and in this way what about Csharp?

The key difference between **compiled** and **interpreted** languages lies in how the code is executed:

Compiled Languages

1. **Definition:**
    - Code written in compiled languages is transformed into machine code (binary code) by a **compiler** before execution.
    - The resulting machine code is directly executed by the CPU.
2. **Characteristics:**
    - **Performance:** Typically faster than interpreted languages because the code is pre-compiled into machine code.
    - **Error Detection:** Compilation catches many syntax and type errors before execution.
    - **Distribution:** The compiled binary can be run on any compatible system without requiring the source code.
    - **Examples:** C, C++, Rust.
3. **Process:**
    - Source Code → Compiler → Machine Code → Execution

Interpreted Languages

1. **Definition:**
    - Code written in interpreted languages is executed line-by-line by an **interpreter** at runtime.
2. **Characteristics:**
    - **Performance:** Generally slower because the interpreter processes the code during execution.
    - **Flexibility:** Easier to debug and modify since there is no compilation step.
    - **Portability:** Source code can run on any platform with the appropriate interpreter.
    - **Examples:** Python, JavaScript, Ruby.
3. **Process:**
    - Source Code → Interpreter → Execution (Line by Line)

C#: Compiled or Interpreted?

C# is **neither purely compiled nor purely interpreted**. It is considered a **hybrid** because it uses a combination of both approaches:

1. **Compilation to Intermediate Language (IL):**
    - C# code is first compiled by the **C# compiler (csc)** into an **Intermediate Language (IL)**, also known as Microsoft Intermediate Language (MSIL).
    - This IL is platform-independent and is stored in an assembly file (e.g., .exe or .dll).
2. **Just-In-Time (JIT) Compilation:**
    - At runtime, the **.NET runtime (CLR - Common Language Runtime)** translates the IL into **native machine code** using a Just-In-Time (JIT) compiler.
    - This approach combines the advantages of portability (via IL) and performance (via JIT compilation).

3- Compare between implicit, explicit, Convert and parse casting?

1. Implicit Casting

- **Definition:** Automatically converts a value from a smaller or compatible data type to a larger or compatible data type. No data loss occurs.
- **Characteristics:**
    - No special syntax or method required.
    - Happens implicitly without programmer intervention.
    - **Only works when there is no risk of data loss.**
- **Examples:**

```
int intVal = 100;

double doubleVal = intVal; // Implicit casting (int to double)
```

- **Limitations:**
    - Only works for compatible types (e.g., int to float, char to int).
    - Cannot be used for incompatible types like string to int.

2. Explicit Casting

- **Definition:** Requires manual intervention to convert a value from one data type to another. Often used when there is potential for data loss.
- **Characteristics:**

- Performed using a cast operator (type).
- **May result in runtime errors** (e.g., data loss, overflow).

- **Examples:**

```
double doubleVal = 123.45;

int intVal = (int)doubleVal; // Explicit casting (double to int)

Console.WriteLine(intVal);  // Output: 123 (fractional part is lost)
```

- **Limitations:**
  - Used for conversions between compatible types.
  - Requires caution to avoid runtime errors.

3. Convert Class

- **Definition:** Provides methods to convert a value from one type to another, even between incompatible types.
- **Characteristics:**
  - Handles **null values** safely (e.g., converting null to 0 for numbers).
  - Includes methods like Convert.ToInt32(), Convert.ToDouble(), etc.
  - **Performs type-safe conversions.**
- **Examples:**

```
string strVal = "123";

int intVal = Convert.ToInt32(strVal); // Converts string to int

Console.WriteLine(intVal); // Output: 123

string nullString = null;

int zero = Convert.ToInt32(nullString); // Safely converts null to 0
```

- **Limitations:**
  - May throw exceptions for invalid formats (e.g., FormatException).
  - Slower compared to direct casting due to additional checks.

---

4. Parse Method

- **Definition:** Converts a string representation of a value to a specific type. Limited to numeric and date types.
- **Characteristics:**

- Commonly used for parsing strings into numeric or date types.
- Throws exceptions if the string is null, empty, or improperly formatted.

- **Examples:**

```
string strVal = "123";
int intVal = int.Parse(strVal); // Converts string to int
Console.WriteLine(intVal); // Output: 123
```

- **Limitations:**
  - **Cannot handle null values** (throws ArgumentNullException).
  - Throws FormatException for invalid strings.